THE OHIO STATE UNIVERSITY

High-Level Connectionist Models

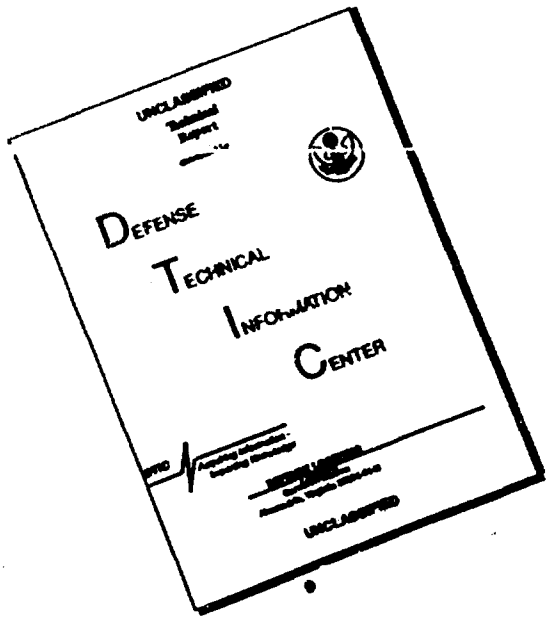
Jordan B. Pollack
Laboratory for AI Research
Department of Computer and Information Science

Department of the Navy
Office of Naval Research
Arlington, Virginia 22217-5660

Grant No. N00014-93-1-0059
Semi-Annual Report

April 1993

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

THE OHIO STATE UNIVERSITY

High-Level Connectionist Models

Jordan B. Pollack
Laboratory for AI Research
Department of Computer and Information Science

Department of the Navy
Office of Naval Research
Arlington, Virginia 22217-5660

Grant No. N00014-93-1-0059
Semi-Annual Report
RF Project No. 760388/726944

April 1993

High Level Connectionist Models

N00014-93-1-0059
Semi annual Report
April 1993

Jordan B. Pollack
Laboratory for AI Research
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210-1277

1.0 Introduction

The high-level connectionist project is concerned with the question of how the representations and processes necessary for high-level symbolic tasks can be achieved within the iterative and numeric style of computing supplied by neural networks. Our final year this project has focused on the question of *modularity*. Traditionally, connectionist networks are treated as a whole – information is dispersed throughout the weights of the network, and the resulting distributed system leads to smooth degradation, etc. Unfortunately, the lack of modularity in such networks also prevents scaling.

Other connectionist researches have realized this problem, and responded by exploring various connectionist architectures for modularity (e.g., Jacobs, Jordan, and Barto, 1990; Nowlan & Hinton, 1991). In these works, however, the modularity is prespecified in terms of a fixed network architecture, which depends on a centralized gating network.

An alternative approach to modularity is found in the design of autonomous robots, a historically nontrivial control task. Brooks (1986, 1991) offers a task-based subsumptive architecture which has achieved some impressive results. However, since machine learning is not up to the task of evolving these systems, engineers of artificial animals have embedded themselves in the design loop as the learning algorithm, and thus all components of the system, as well as their interactions, must be carefully crafted by the engineer (see, e.g., Connell, 1990).

Research aimed at replacing the engineer in these systems is at an early stage. For example, Maes (1991) proposes an Agent Network Architecture which allows a modular agent to learn to satisfy goals such as "relieve thirst"; however, she presumes detailed high-level modules (such as "pick-up-cup" and "bring-mouth-to-cup"), and her system

learns only the connections between these modules. A better approach would be to let the modules and connectionist develop automatically in response to the demands of the task.

2.0 Progress & Highlights

By combining Brooks' ideas of subsumption with traditional connectionist models of modularity (Jacobs, Jordan, and Barto, 1990), we have developed a novel architecture in which new modules can be added and trained with minimal disturbance to existing connections. The result is ADDAM (or ADDitive ADaptive Modules), a modular connectionist agent whose behavioral repertoire evolves as the complexity of the environment is increased. When placed in a simulated world of ice, food, and blocks, ADDAM exhibits complex behaviors due to the interactions of its simple modules, as shown in Figure 1 (Saunders et al., 1992).

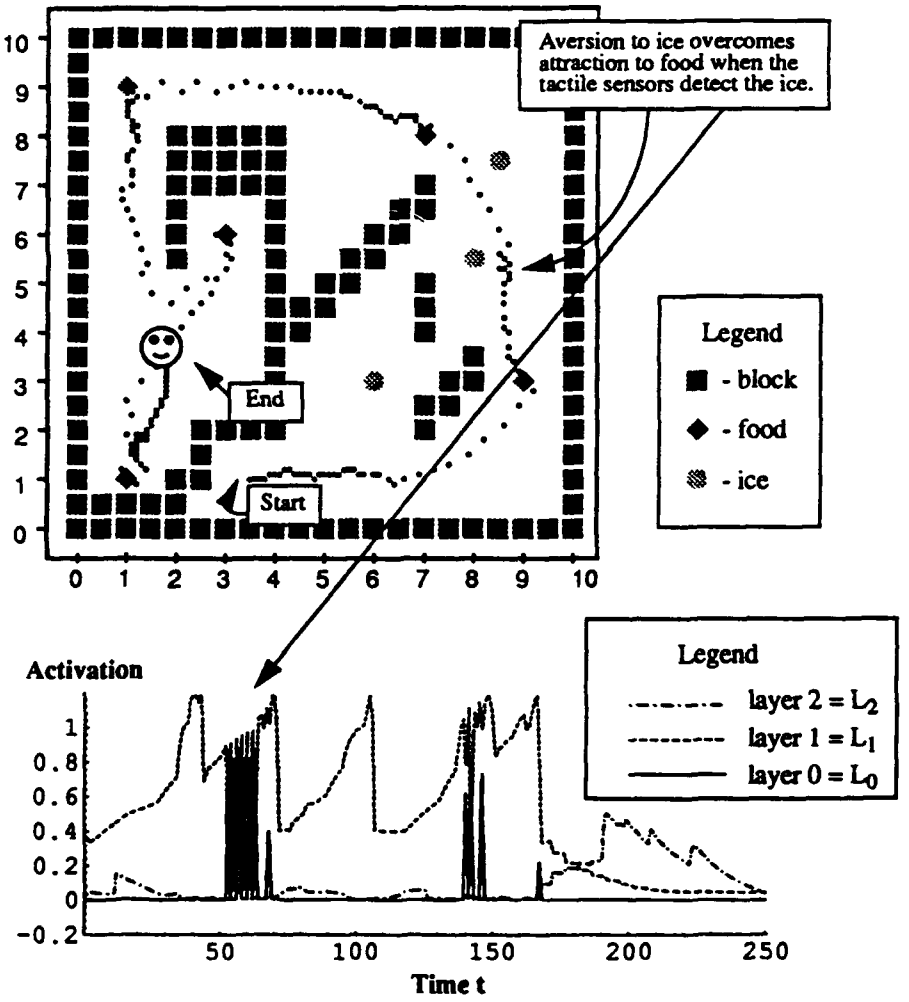


Figure 1: Addam's emergent behavior in a complex environment, with graph showing the activations of layers 0, 1, and 2.

There is a distinct methodological difference between this work and that of Brooks. In creating his agents, Brooks first performs a behavioral decomposition, but in implementing each layer, he performs a functional decomposition of the type he himself warns against (Brooks, 1991, p. 146). In training Addam, on the other hand, we first perform a behavioral decomposition, and then let backpropagation decompose each behavior appropriately. This automation significantly lessens the arbitrary nature of behavior-based architectures which has thus far limited the import of Brooks' work to cognitive science.

Our next goal was to take this one step further, by automating the behavioral decomposition. In other words, we desired to have the modules evolve in response to the demands of the task. To accomplish this, we needed a training mechanism more robust than backpropagation, so we turned towards *genetic algorithms* (GAs). These algorithms, based on principles adopted from natural selection, allow solutions to be evolved which fit the requirements of an environment.

There is an extensive body of work applying GAs to evolving neural networks, but most simply use GAs to set the weights for a fixed-structure network. Those that attempt to evolve network structure do so in a very limited way. (See Schaffer, et al., 1992 for a good overview.) Thus before applying GAs to network modularization, we first had to solve the "generalized network acquisition" problem, i.e., the problem of acquiring both network structure and weight values simultaneously. The result was GNARL, an algorithm for *GeNeralized Acquisition of Recurrent Links* (Angeline, Saunders, and Pollack, 1993).

The power of GNARL is shown the *Tracker* task, described by Jefferson, et al. (1991). In this problem, a simulated ant is placed on a two-dimensional toroidal grid that contains a trail of food. The ant traverses the grid, eating in one time step any food it contacts. The goal of the task is to maximize the number of pieces of food the ant eats within a predefined allotted time. The trail of food used in the experiment (shown in Figure 2a)

was hand-crafted in the original study to be increasingly difficult for the evolving ants.

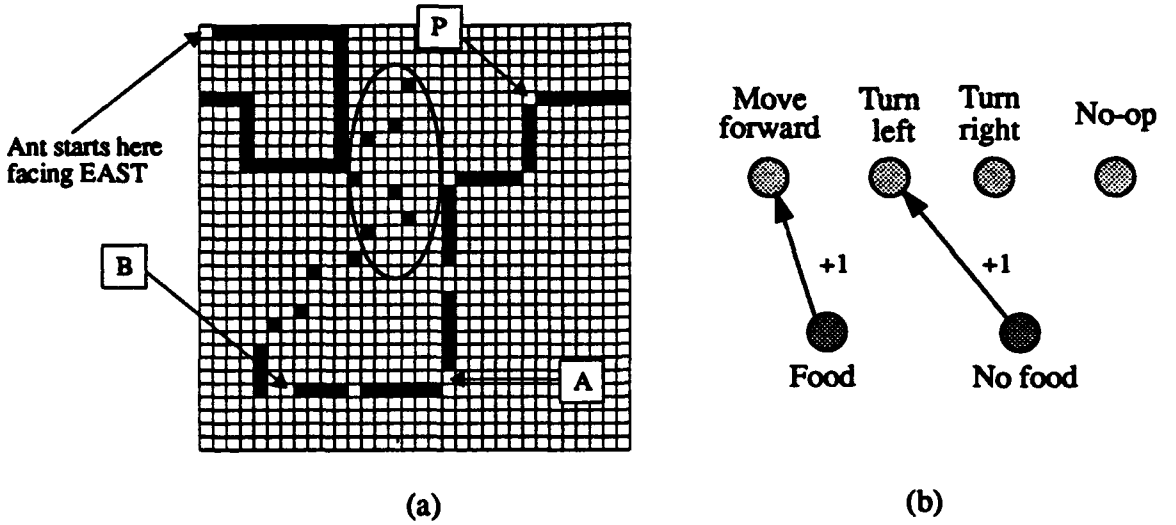


Figure 2: The ant problem. (a) The trail is connected initially, but becomes progressively more jagged and winding, hence more difficult to follow. The underlying 2-d grid is toroidal, so that position "P" is the first break in the trail. Positions "A" & "B" indicate the two positions where the network of our first run behaves differently from the simple FSA Jefferson et al. hand-crafted to perform this task. The ellipse indicates the 7 pieces of food that the network of the second run failed to reach. (b) The semantics of the I/O units for the ant network. This simple network "eats" 42 pieces of food before spinning endlessly in place at position P, illustrating a very deep local minimum in the search space.

Following Jefferson, *et al* (1991), the ant is controlled by a network with two input nodes and four output nodes, as shown in Figure 2b. The first input node denotes the presence of food in the square directly in front of the ant; the second denotes the absence of food in this same square, restricting the possible legal inputs to the network to (1, 0) or (0, 1). Each of the four output units corresponds to a unique executable action – move forward, turn left, turn right, or no-op. Each ant is in an implicit sense/act loop that repeatedly sets the input activations of the network, computes the activations of the output nodes, determines the output node with maximum activation and executes its associated action. Every application of the sense/act loop is assumed to happen in a single time step. Once a position with food is visited, the food is removed. The fitness function used in this task is simply the number of grid positions cleared in 200 time steps.

In these experiments, we used a population of 100 networks. In the first run (2090 generations using 104,600 network evaluations), GNARL found a network (shown in Figure 3b) that cleared 81 grid positions within the 200 time steps. When we allowed this ant to run for an additional 119 time steps, it successfully cleared the entire trail.

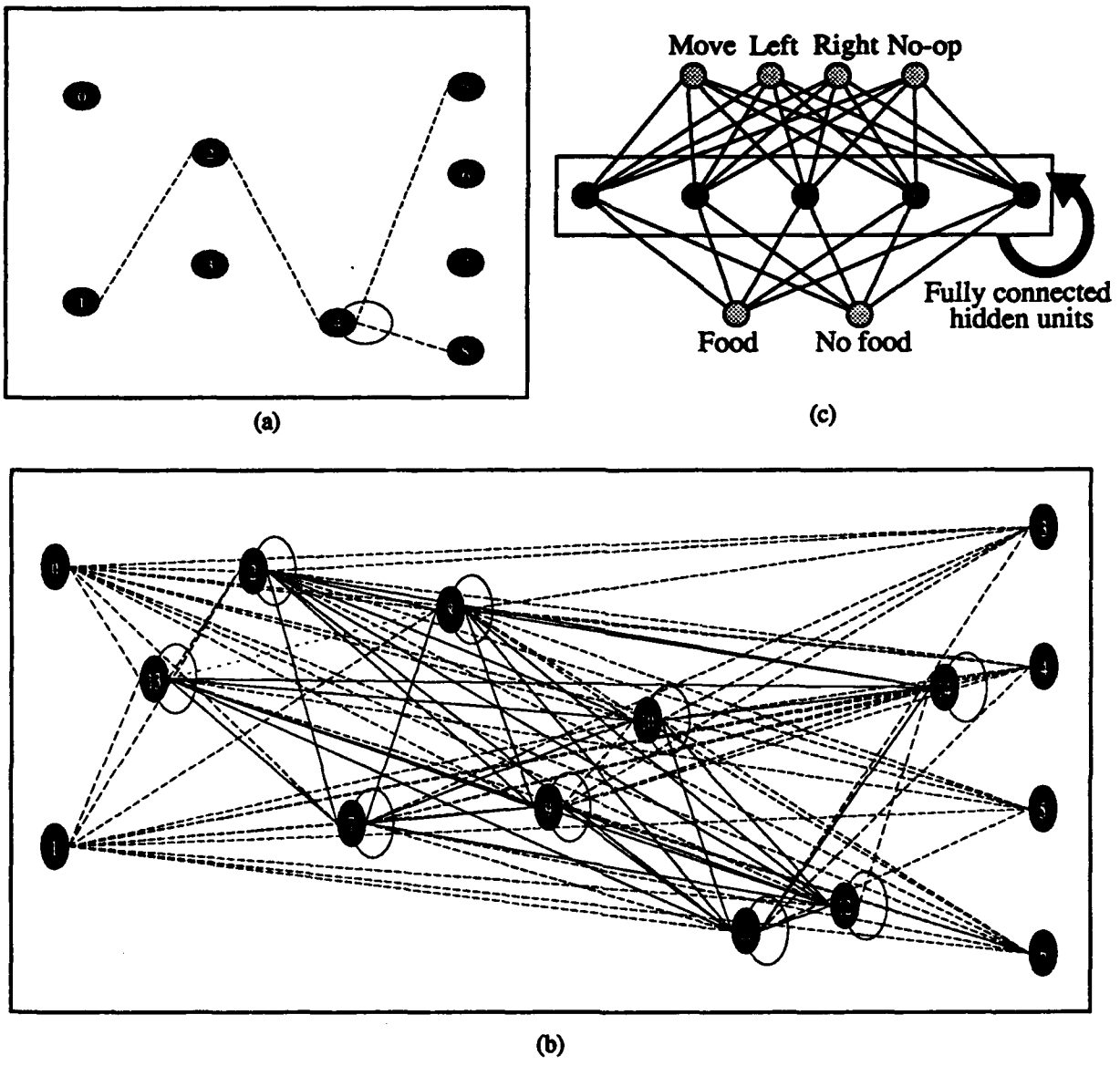


Figure 3: The Tracker Task, *first run. (a) The best network in the initial population. Nodes 0 & 1 are input, nodes 5-8 are output, and nodes 2-4 are hidden node. (b) Network evolved by GNARLY after 2090 generations. Forward links are dashed; bidirectional links & loops are solid. The single backlink (from node 8 to 13) is dashed, but lighter than the others. This network clears the trail in 319 epochs. (c) Jefferson et al.'s fixed network structure for the Tracker task.*

3.0 Summary and Conclusion

With the success of GNARL at generalized network acquisition, we are now poised to return to the question of modularity. Although GAs are inherently nonmodular, previous work in our lab has explored extending the capabilities of these algorithms through modularization (Angeline & Pollack, 1992a, 1992b). Springboarding from that work, we plan on developing a modular version of GNARL, one that will freeze subsets of nodes and weights during the evolution of the network. The modules developed by

this process will emerge in response to the demands of the task, and be free from the bias of user-specification that permeates other work in connectionist modularity.

4.0 References

Angeline, P. A., and Pollack, J. B. (1992a). The evolutionary induction of subroutines. *Proceedings of the 14th Annual Conference of the Cognitive Science Society,* Bloomington.

Angeline, P. and Pollack, J. (1992b). Coevolving high-level representations. LAIR Technical Report 92-PA-COEVOLVE, The Ohio State University, Columbus Ohio. To appear to Artificial Life III.

Angeline, P., Saunders, G., Pollack, J. (1993). An evolutionary algorithm that constructs recurrent neural networks. LAIR Technical Report 93-PA-GNARL, The Ohio State University, Columbus Ohio. Submitted to *IEEE Transactions on Neural Networks: Special Issue on Evolutionary Programming.*

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation,* 2(1):14–23.

Brooks, R. A. (1991). Intelligence without representations. *Artificial Intelligence,* 47:139–159.

Connell, J. H. (1990). Minimalist Mobile Robotics: A Colony-style Architecture for an Creature, Volume 5 of Perspectives in Artificial Intelligence. Academic Press, San Diego.

Jacobs, R. A., Jordan, M. I., and Barto, A. G. (1990). Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks. Cognitive Science, 15:219-250.

Jefferson, D., Collins, R., Cooper, C., Dyer, M., Flowers, M., Korf, R., Taylor, C., and Wang, A. (1991). Evolution as a theme in artificial life: The genesys/tracker system. In Langton, C. G., Taylor, C., Farmer, J. D., and Rasmussen, S., editors, *Artificial LIfe II: Proceedings of the Workshop on Artificial Life.* pages 549–577. Addison-Wesley.

Maes, P. (1991). The agent network architecture. In AAAI Spring Symposium on Integrated Intelligent Architectures, March.

Nowlan, S. J. and Hinton, G. E. (1991). Evaluation of adaptive mixtures of competing experts. In Lippmann, R., Moody, J., and Touretzky, D., editors, Advances in Neural Information Processing 3. Morgan Kaufmann, pages 774-780.

Saunders, G. M., Kolen, J. F., Angeline, P. A., and Pollack, J. B. (1992). Additive modular learning in preemptrons. *Proceedings of the 14th Annual Conference of the Cognitive Science Society,* Bloomington.

Schaffer, J. D., Whitley, D., and Eshelman, L. J. (1992). Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Proceedings of COGANN-92 International Workshop on Combinations of Genetic Algorithms and Neural Networks.*

Appendices

Artificial Intelligence 59 (1993) 355-369
Elsevier

ARTINT 1014

Book Review

On wings of knowledge: a review of Allen Newell's *Unified Theories of Cognition* *

Jordan B. Pollack

Laboratory for AI Research, The Ohio State University, 2036 Neil Avenue, Columbus, OH 43210, USA

Received January 1992
Revised July 1992

1. Introduction

Besides being a status report on the Soar project, *Unified Theories of Cognition* is Allen Newell's attempt at directing the field of cognitive science by example. Newell argues that his approach to "unification", which involves the programmed extension of a single piece of software-architecture-as-theory to as many psychological domains as possible, is the proper research methodology for cognitive science today:

> In this book I'm not proposing Soar as *the* unified theory of cognition. Soar is, of course, an interesting candidate. With a number of colleagues I am intent on pushing Soar as hard as I can to make it into a viable unified theory. But my concern here is that cognitive scientists consider working with *some* unified theory of cognition. Work with ACT*, with CAPS, with Soar, with CUTC, a connectionist unified theory of cognition. Just work with some UTC. (p. 430)

Correspondence to: J.B. Pollack, Laboratory for AI Research, The Ohio State University, 2036 Neil Avenue, Columbus, OH 43210, USA. E-mail: pollack@cis.ohio-state.edu

* (Harvard University Press, Cambridge, MA, 1990); 549 pages

Over the past decade, Newell and his colleagues at numerous universities (including my own) have applied Soar to a number of different domains, and have adopted a goal of making it toe the line on psychological results. This is a very ambitious goal, and Newell knows it:

> The next risk is to be found guilty of the sin of presumption. Who am I, Allen Newell, to propose a unified theory of cognition ... Psychology must wait for its Newton. (p. 37)

Newell is clearly entitled by a life of good scientific works to write a book at such a level and, in my opinion, it is the most substantial and impressive, by far, of recent offerings on the grand unified mind. My entitlement to review his book is less self-evident, however—who am I to stand in judgement over one of the founding fathers of the field? And so I fear I am about to commit the sin of presumption as well, and to compound it, moreover, with the sin of obliqueness: Because my argument is not with the quality of Newell's book, but with the direction he is advocating for cognitive science, I will not review his theory in detail. Rather, I will adopt a bird's eye view and engage only the methodological proposal. I will, however, belabor one small detail of Newell's theory, its name, and only to use as my symbolic launching pad.

2. Artificial intelligence and mechanical flight

The origin of the name Soar, according to high-level sources within the project, was originally an acronym for three primitive components of the problem-space method. But shortly, these components were forgotten, leaving the proper noun in their stead, a name which evokes "grand and glorious things", and also puts us in mind of the achievement of mechanical flight, AI's historical *doppelgänger*.

> Among those who had worked on the problem [of mechanical flight] I may mention [da Vinci, Cayley, Maxim, Parsons, Bell, Phillips, Lilienthal, Edison, Langley] and a great number of other men of ability. But the subject had been brought into disrepute by a number of men of lesser ability who had hoped to solve the problem through devices of their own invention, which had all of themselves failed, until finally the public was lead to believe that flying was as impossible as perpetual motion. In fact, scientists of the standing of Guy Lussac ... and Simon Newcomb ... had attempted to prove it would be impossible to build a flying machine that would carry a man. (Wright [25, p. 12] [1]

[1] This book is a reissued collection of essays and photographs about the Wright's research and development process. It includes three essays by Orville Wright, and two interpretive essays by Fred C. Kelly. Subsequent citations are to this edition.

I will leave the substitution of contemporary scientists to the reader. Simply put, the analogy "Airplanes are to birds as smart machines will be to brains", is a widely repeated AI mantra with several uses. One is to entice consumers by reminding them of the revolution in warfare, transportation, commerce, etc. brought about by mechanical flight. Another is to encourage patience in those same consumers by pointing to the hundreds of years of experimental work conducted before the success of mechanical flight! A third is to chant it, eyes closed, ignoring the complex reality of biological mechanism.

Although it is quite likely that the analogy between AI and mechanical flight arose spontaneously in the community of AI pioneers, its earliest written appearance seems to be in a "cold war for AI" essay by Paul Armer, then of the Rand Corporation, in the classic collection *Computers and Thought*:

> While it is true that Man wasted a good deal of time and effort trying to build a flying machine that flapped its wings like a bird, the important point is that it was the understanding of the law of aerodynamic lift (even though the understanding was quite imperfect at first) over an airfoil which enabled Man to build flying machines. A bird isn't sustained in the air by the hand of God—natural laws govern its flight. Similarly, natural laws govern what [goes on inside the head]. Thus I see no reason why we won't be able to duplicate in hardware the very powerful processes of association which the human brain has, once we understand them. (Armer [1, p.398])

We all agree that once we understand how natural law governs what goes on in the head, we will be able to mechanize thought, and will then have the best scientific theory of cognition, which could be refined into the technology for "general intelligence". But our field has basically ignored natural law, and settled comfortably upon methodologies and models which involve only the perfect simulation of arbitrary "software" laws. I believe that we are failing to integrate several key principles which govern cognition and action in biological and physical systems, and that the incorporation of these should be the priority of cognitive science rather than of the writing of large programs.

3. Deconstructing the myths of mechanical flight

There are two myths in Armer's analogy which are important to correct. The first is that flight is based mainly upon the principle of the airfoil. The second is that the mechanical means by which nature solved the problem are

irrelevant. Translating through the analogy, these two myths are equivalent to believing that cognition is based mainly upon the principle of universal computation, and that the mechanical means by which nature solved the problem are irrelevant.

Although my favorite sections of Newell's book are those in which he emphasizes the importance of constraints from biology and physics, he conducts his research in a way which is consistent with the myths. Indeed the myths are principally supported by his arguments, both the Physical Symbol System Hypothesis [17], which is the assertion that Universal Computation is enough, and the Knowledge Level Hypothesis [16], which legitimizes theories involving only software laws, even though their very existence is based only upon introspection.

In order to see why these myths are a stumbling block to the achievement of mechanical cognition, I will examine several aspects of the solution to mechanical flight, using the reports by Orville Wright.

3.1. The airfoil principle

The principle of aerodynamic lift over an airfoil was around for hundreds of years before the advent of mechanical flight. The Wright brothers just tuned the shape to optimize lift:

> The pressures on squares are different from those on rectangles, circles, triangles or ellipses; arched surfaces differ from planes, and vary among themselves according to the depth of curvature; true arcs differ from parabolas, and the latter differ among themselves; thick surfaces from thin ... the shape of an edge also makes a difference, so thousands of combinations are possible in so simple a thing as a wing .... Two testing machines were built [and] we began systematic measurements of standard surfaces, so varied in design as to bring out the underlying causes of differences noted in their pressures (Wright [25, p. 84])

Assume that the principle of Universal Computation is to AI what the principle of aerodynamic lift is to mechanical flight. In Chapter 2 of this book, Newell reiterates, in some detail, the standard argument for the status quo view of cognition as symbolic computation:

- Mind is flexible, gaining power from the formation of "indefinitely rich representations" and an ability to compose transformations of these representations. (pp. 59–63).
- Therefore mind must be a universal symbol processing machine (pp. 70–71).
- It is believed that most universal machines are equivalent (p. 72).

If one holds that the flexibility of the mind places it in the same class as the other universal machines (subject to physical limits, of course), then the mathematics tells us we can use any universal computational model for describing mind or its subparts (and the biological path is irrelevant). So, for example, any of the four major theories of computation developed (and unified) this century—Church's lambda calculus, Post's production system, Von Neumann's stored program machine, and universal automata (e.g. Turing's Machine)—could be used as a basis for a theory of mind. Of course, these theories were too raw and difficult to program, but have evolved through human ingenuity into their modern equivalents, each of which is a "Universal Programming Language" (UPL): LISP, OPS5, C, and ATN's, respectively.

If we plan to express our UTCs in UPLs, however, we must have a way to distinguish between these UPLs in order to put some constraints on our UTCs, so that they aren't so general as to be vacuous. There are at least five general strategies to add constraints to a universal system: architecture, tradition, extra-disciplinary goals, parsimony, and ergonomics.

The first way to constrain a UPL is to build something, anything, on top of it, which constrains either what it can compute, how well it can compute it, or how it behaves while computing it. This is called architecture, and Newell spends pages 82–88 discussing architecture in some detail. In a universal system, one can build architecture on top of architecture and shift attention away from the basic components of the system to arbitrary levels of abstraction.

The second method of constraining a universal theory is by sticking to "tradition", the practices whose success elevates them to beliefs handed down orally though the generations of researchers. Even though any other programming language would serve, the tradition in American AI is to build systems from scratch, using LISP, or to build knowledge into production systems. Newell is happy to represent the "symbolic cognitive psychology" tradition (p. 24) against the paradigmatic revolutions like Gibsonianism [10] or PDPism [21].

A third way we can distinguish between competing universals is to appeal to scientific goals outside of building a working program. We can stipulate, for example, that the "most natural" way to model a phenomenon in the UPL must support extra-disciplinary goals. Algorithmic efficiency, a goal of mainstream computer science, can be used to discriminate between competing models for particular tasks. Robust data from psychological experiments can be used to discriminate among universal theories on the basis of matching an implemented system's natural behavior with the observed data from humans performing the task. Finally, as a subset of neural network researchers often argue, the model supporting the theory must be "biologically correct". Newell, of course,

relies very heavily on the psychological goals to justify his particular UPL.

Fourth, parsimony can be called into play. Two theories can be compared as to the number of elements, parameters, and assumptions needed to explain certain phenomena, and the shorter one wins. Newell makes a good point that the more phenomena one wishes to explain, the more benefit is gained from a unified theory, which ends up being shorter than the sum of many independent theories. This is called "amortization of theoretical constructs" (p. 22) and is one of Newell's main arguments for why psychologists ought to adopt his unified theory paradigm for research. However, such a unification can be politically difficult to pull off when different subdisciplines of a field are already organized by the parsimony of their own subtheories.

Fifth, we might be able to choose between alternatives on the basis of ergonomics. We can ask the question of programmability. How easy is it to extend a theory by programming? How much work is it for humans to understand what a system is doing? The Turing Machine model "lost" to the stored program machine due to the difficulty of programming in quintuples. Systems which use explicit rules rather than implicit knowledge-as-code are certainly easier to understand and debug, but may yield no more explanatory power.

However, unless the constraints specifically strip away the universality, such that the cognitive theory becomes *an application of* rather than *an extension to* the programming language, the problem of theoretical underconstraint remains. Following Newell's basic argument, one could embark on a project of extensively crafting a set of cognitive models in any programming language, say C++, matching psychological regularities, and reusing subroutines as much as possible, and the resultant theory would be as predictive as Soar:

> Soar does not automatically provide an explanation for anything just because it is a universal computational engine. There are two aspects to this assertion. First from the perspective of cognitive theory, Soar has to be universal, because humans themselves are universal. To put this the right way around—Soar is a universal computational architecture; therefore it predicts that the human cognitive architecture is likewise universal. (p. 248)

Thus, just as the Wright brothers discovered different lift behaviors in differently shaped airfoils, cognitive modelers will find different behavioral effects from different universal language architectures. The principle of the airfoil was around for hundreds of years, and yet the key to mechanical flight did not lie in optimizing lift. Using a UPL which optimizes "cognitive lift" is not enough either, as practical issues of scale and control will still assert themselves.

3.2. Scaling laws

> Our first interest [in the problem of flight] began when we were children. Father brought home to us a small toy actuated by a rubber [band] which would lift itself into the air. We built a number of copies of this toy, which flew successfully, but when we undertook to build a toy on a much larger scale it failed to work so well. (Wright [25, p. 11])

The youthful engineers did not know that doubling the size of a model would require eight times as much power. This is the common feeling of every novice computer programmer hitting a polynomial or exponential scaling problem with an algorithm. But there were many other scaling problems which were uncovered during the Wrights' mature R&D effort, beyond mere engine size:

> We discovered in 1901 that tables of air pressures prepared by our predecessors were not accurate or dependable. (Wright [25, p. 55])

> We saw that the calculations upon which all flying-machines had been based were unreliable, and that all were simply groping in the dark. Having set out with absolute faith in the existing scientific data, we were driven to doubt one thing after another .... Truth and error were everywhere so intimately mixed as to be indistinguishable. (Wright [25, p. 84])

Thus it is not unheard of for estimates of scaling before the fact to be way off, especially the first time through based upon incomplete scientific understanding of the important variables. Estimates of memory size [12,13] for example, or the performance capacity of a brain [15,24] may be way off, depending on whether memory is "stored" in neurons, synapses, or in modes of behaviors of those units. So the number of psychological regularities we need to account for in a UTC may be off:

> Thus we arrive at about a third of a hundred regularities about [typing] alone. Any candidate architecture must deal with most of these if it's going to explain typing .... Of course there is no reason to focus on typing. It is just one of a hundred specialized areas of cognitive behavior. It takes only a hundred areas at thirty regularities per area to reach the ~~3000 total regularities cited at the beginning of this chapter .... Any architecture, especially a candidate for a unified theory of cognition, must deal with them all—hence with thousands of regularities. (p. 243)

There is a serious question about whether thousands of regularities are enough, and Newell recognizes this:

> In my view its it time to get going on producing unified theories of cognition—before the data base doubles again and the number of visible clashes increases by the square or cube. (p. 25)

While Newell estimates 3000 regularities, my estimate is that the number of regularities is unbounded. The "psychological data" industry is a generative system, linked to the fecundity of human culture, which Newell also writes about lucidly:

> What would impress [The Martian Biologist] most is the efflorescence of adaptation. Humans appear to go around simply creating opportunities of all kinds to build different response functions. Look at the variety of jobs in the world. Each one has humans using different kinds of response functions. Humans invent games. They no sooner invent one game than they invent new ones. They not only invent card games, but they collect them in a book and publish them .... Humans do not only eat, as do all other animals, they prepare their food ... inventing [hundreds and thousands of] recipes. (p. 114)

Every time human industry pops forth with a new tool or artifact, like written language, the bicycle, the typewriter, Rubik's cube, or rollerblades, another 30 regularities will pop out, especially if there is a cost justification to do the psychological studies, as clearly was the case for typing and for reading. This is not a good situation, especially if programmers have to be involved for each new domain. There will be a never-ending software battle just to keep up:

> Mostly, then, the theorist will load into Soar a program (a collection of productions organized into problem spaces) of his or her own devising...The obligation is on the theorist to cope with the flexibility of human behavior in responsible ways. (Newell, p. 249)

If the road to unified cognition is through very large software efforts, such as Soar, then we need to focus on scalable control laws for software.

3.3. Control in high winds

Although we might initially focus on the scaling of static elements like wing span and engine power, *to duplicate the success of mechanical flight, we should focus more on the scaling of control.* For the principal contribution of the Wright brothers was not the propulsive engine, which had its own

economic logic (like the computer), nor the airfoil, a universal device they merely tuned through experiments, but their insight about how to control a glider when scaled up enough to carry an operator:

> Lilienthal had been killed through his inability to properly balance his machine in the air. Pilcher, an English experimenter had met with a like fate. We found that both experimenters had attempted to maintain balance merely by the shifting of the weight of their bodies. Chanute and all the experimenters before 1900, used this same method of maintaining the equilibrium in gliding flight. We at once set to work to devise a more efficient means of maintaining the equilibrium .... It was apparent that the [left and right] wings of a machine of the Chanute double-deck type, with the fore-and-aft trussing removed, could be warped ... in flying ... so as to present their surfaces to the air at different angles of incidences and thus secure unequal lifts .... (Wright [25, p. 12])

What they devised, and were granted a monopoly on, was the *Aileron principle*, the general method of maintaining dynamical equilibrium in a glider by modifying the shapes of the individual wings, using cables, to provide different amounts of lift to each side. (It is not surprising that the Wright brothers were bicycle engineers, as this control principle is the same one used to control two wheeled vehicles—iterated over-correction towards the center.)

Translating back through our analogy, extending a generally intelligent system for a new application by human intervention in the form of programming is "seat of the pants" control, the same method that Lilienthal applied to maintaining equilibrium by shifting the weight of his body.

Just as the source of difficulty for mechanical flight was that scaling the airfoil large enough to carry a human overwhelmed that human's ability to maintain stability, the source of difficulty in the software engineering approach to unified cognition is that scaling software large enough to explain cognition overwhelms the programming teams' ability to maintain stability.

It is well known that there are limiting factors to software engineering [5], and these limits could be orders of magnitude below the number of "lines of code" necessary to account for thousands of psychological regularities or to achieve a "general intelligence". Since software engineers haven't figured out how to build and maintain programs bigger than 10–100 million lines of code, why should people in AI presume that it can be done as a matter of course? [7].

What is missing is some control principle for maintaining dynamical coherence of an ever-growing piece of software in the face of powerful winds

of change. While I don't pretend to have the key to resolving the software engineering crisis, I believe its solution may rest with building systems from the bottom up using robust and stable cooperatives of goal-driven modules locked into long-term prisoner's dilemmas [2], instead of through the centralized planning of top-down design. The order of acquisition of stable behaviors can be very important to the solution of hard problems.

3.4. On the irrelevancy of flapping

> Learning the secret of flight from a bird was a good deal like learning the secret of magic from a magician. After you know the trick and know what to look for, you see things that you did not notice when you did not know exactly what to look for. (Wright (attributed) [25, p. 5])

When you look at a bird flying, the first thing you see is all the flapping. Does the flapping explain how the bird flies? Is it reasonable to theorize that flapping came first, as some sort of cooling system which was recruited when flying became a necessity for survival? Not really, for a simpler explanation is that most of the problem of flying is in finding a place within the weight/size dimension where gliding is possible, and getting the control system for dynamical equilibrium right. Flapping is the last piece, the propulsive engine, but in all its furiousness, it blocks our perception that the bird first evolved the aileron principle. When the Wrights figured it out, they saw it quite clearly in a hovering bird.

Similarly, when you look at cognition, the first thing you see is the culture and the institutions of society, human language, problem solving, and political skills. Just like flapping, symbolic thought is the last piece, the engine of social selection, but in all its furiousness it obscures our perception of cognition as an exquisite control system competing for survival while governing a very complicated real-time physical system.

Once you get a flapping object, it becomes nearly impossible to hold it still enough to retrofit the control system for equilibrium. Studying problem solving and decision making first because they happen to be the first thing on the list (p. 16) is dangerous, because perception and motor control may be nearly impossible to retrofit into the design.

This retrofit question permits a dissection of the "biological correctness" issue which has confounded the relationship between AI and connectionism. The naive form, which applied to work in computational neuroscience but not to AI, is "ontogenetic correctness", the goal of constructing one's model with as much neural realism as possible. The much deeper form, which could be a principle someday, is "phylogenetic correctness", building a model

which could have evolved bottom up, without large amounts of arbitrary top-down design. Phylogenetically correct systems acquire their behaviors in a bottom-up order that could, theoretically, recapitulate evolution or be "reconverged" upon by artificial evolution. Thus, while the airplane does not flap or have feathers, the Wrights' success certainly involved "recapitulation" of the phylogenetic order of the biological invention of flight: the airfoil, dynamical balance, and then propulsion.

4. Physical law versus software law

By restricting ourselves to software theories only, cognitive scientists might be expending energy on mental ornithopters. We spin software and logic webs endlessly, forgetting every day that there is no essential difference between Fortran programs and LISP programs, between sequential programs and production systems, or ultimately, between logics and grammars. All such theories rely on "software law" rather than the natural law of how mechanisms behave in the universe.

Software laws, such as rules of predicate logic, may or may not have existed before humans dreamed them up. And they may or may not have been "implemented" by minds or by evolution. What is clear is that such laws can be created *ad infinitum*, and then simulated and tested on our physical symbol systems: The computer simulation is the sole guaranteed realm of their existence.

An alternative form of unification research would be to *unify cognition with nature*. In other words, to be able to use the same kind of natural laws to explain the complexity of form and behavior in cognition, the complexity of form and behavior in biology, and the complexity of form and behavior in inanimate mechanisms.

I realize this is not currently a widely shared goal, but by applying Occam's razor to "behaving systems" on all time scales (p. 152) why not use the ordinary equations of physical systems to describe and explain the complexity and control of all behavior? In an illuminating passage, Newell discusses control theory:

> To speak of the mind as a controller suggests immediately the language of control systems—of feedback, gain, oscillation, damping, and so on. It is a language that allows us to describe systems as purposive. But we are interested in the full range of human behavior, not only walking down a road or tracking a flying bird, but reading bird books, planning the walk, taking instruction to get to the place, identifying distinct species, counting the new

> additions to the life list of birds seen, and holding conversations about it all afterward. When the scope of behavior extends this broadly, it becomes evident that the language of control systems is really locked to a specific environment and class of tasks—to continuous motor movement with the aim of pointing or following. For the rest it becomes metaphorical. (p. 45)

I think Newell has it backwards. It is the language of symbol manipulation which is locked to a specific environment of human language and deliberative problem solving. Knowledge level explanations only metaphorically apply to complex control systems such as insect and animal behavior [4,6], systems of the body such as the immune or circulatory systems, the genetic control of fetal development, the evolutionary control of populations of species, cooperative control in social systems, or even the autopoetic control system for maintaining the planet. Certainly these systems are large and complex and have some means of self-control while allowing extreme creativity of behavior, but the other sciences do not consider them as instances of universal computing systems running software laws, divorced from the physical reality of their existence!

We have been led up the garden path of theories expressed in rules and representations because simple mathematical models, using ordinary differential equations, neural networks, feedback control systems, stochastic processes, etc. have for the most part been unable to describe or explain the generativity of structured behavior with unbounded dependencies. especially with respect to language [8]. This gulf between what is needed for the explanation of animal and human cognitive behavior and what is offered by ordinary scientific theories is really quite an anomaly and indicates that our understanding of the principles governing what goes on in the head have been very incomplete.

But where might governing principles for cognition come from besides computation? The alternative approach I have been following over the past several years has emerged from a simple goal to develop neural network computational theories which gracefully admit the generative and representational competance of symbolic models. This approach has resulted in two models [18,19] with novel and interesting behaviors. In each of these cases, when pressed to the point of providing the same theoretical capacity as a formal symbol system, I was forced to interpret these connectionist networks from a new point of view, involving fractals and chaos—a dynamical view of cognition, more extreme than that proposed by Smolensky [23]. I have thus been lead to a very different theoretical basis for understanding cognition, which I will call the "Dynamical Cognition Hypothesis", that:

> The recursive representational and generative capacities required for cognition arise directly out of the complex behavior of nonlinear dynamical systems.

In other words, neural networks are merely the biological implementation level for a computation theory not based upon symbol manipulation, but upon complex and fluid patterns of physical state. A survey of cognitive models based upon nonlinear dynamics is beyond this review [20], however, I can briefly point to certain results which will play an important role in this alternative unification effort.

Research in nonlinear systems theory over the past few decades has developed an alternative explanation for the growth and control of complexity [11]. Hidden within the concept of deterministic "chaotic" systems which are extremely sensitive to small changes in parameters is the surprise that precise tuning of these parameters can lead to the generation of structures of enormous apparent complexity, such as the famous Mandelbrot set [14].

There is a clear link between simple fractals, like Cantor dust, and rewriting systems ("remove the middle third of each line segment"), and Barnsley has shown how such recursive structures can be found in the limit behavior of very simple dynamical systems [3]. The very notion of a system having a "fractional dimension" is in fact the recognition that its apparent complexity is governed by a "power law" [22].

The equations of motion of nonlinear systems are not different in kind from those of simpler physical systems, but the evoked behavior can be very complicated, to the point of appearing completely random. Even so, there are "universal" laws which govern these systems at all scales, involving where and when phase transitions occur and how systems change from simple to complex modes, passing through "critical" states, which admit long-distance dependencies between components.

The logistic map $x_{t+1} = kx_t(1 - x_t)$ is a well-studied example of a simple function iterated over the unit line where changes in k (between 0 and 4) lead to wildly different behaviors, including convergence, oscillation, and chaos. In a fundamental result, Crutchfield and Young have exhaustively analyzed sequences of most significant bits generated by this map[2] and have shown that at critical values of k, such as 3.5699, these bit sequences have unbounded dependencies, and are not describable by a regular grammar, but by an indexed context-free grammar [9].

Without knowing where complex behavior comes from, in the logistic map, in critically tuned collections of neural oscillators, or in the Mandelbrot set, one could certainly postulate a very large rule-based software system,

[2]They analyzed the bit string $y_t = floor(0.5 + x_t)$.

operating omnipotently behind the scenes, like a deity whose hand governs the fate of every particle in the universe.

5. Conclusion

I want to conclude this review with a reminder to the reader to keep in mind the "altitude" of my criticism, which is about the research methodology Newell is proposing based upon the status quo of myths in AI, and not about the detailed contents of the book. These are mature and illuminated writings, and Newell does an excellent job of setting his work and goals into perspective and recognizing the limitations of his theory, especially with respect to the puzzles of development and language.

Despite my disagreement with Newell's direction, I was educated and challenged by the book, and endorse it as an elevator for the mind of all students of cognition. But still, I would warn the aspiring cognitive scientist not to climb aboard any massive software engineering efforts, expecting to fly:

> You take your seat at the center of the machine beside the operator. He slips the cable, and you shoot forward .... The operator moves the front rudder and the machine lifts from the rail like a kite supported by the pressure of the air underneath it. The ground is first a blur, but as you rise the objects become clearer. At a height of 100 feet you feel hardly any motion at all, except for the wind which strikes your face. If you did not take the precaution to fasten your hat before starting, you have probably lost it by this time .... (Wright, [25, p. 86])

References

[1] P. Armer, Attitudes towards artificial intelligence, in: E.A. Feigenbaum and J.A. Feldman, eds., *Computers and Thought* (McGraw Hill, New York, 1963) 389–405.
[2] R. Axelrod, *The Evolution of Cooperation* (Basic Books, New York, 1984).
[3] M.F. Barnsley, *Fractals Everywhere* (Academic Press, San Diego, CA, 1988).
[4] R. Beer, *Intelligence as Adaptive Behavior: An Experiment in Computational Neuroethology* (Academic Press, New York, 1990).
[5] F.P. Brooks, *The Mythical Man-Month* (Addison-Wesley, Reading, MA, 1975).
[6] R. Brooks, Intelligence without representation, *Artif. Intell.* **47** (1–3) (1991) 139–160.
[7] C. Cherniak, Undebuggability and cognitive science, *Commun. ACM* **31** (4) (1988) 402–412.
[8] N. Chomsky, *Syntactic Structures* (Mouton, The Hague, Netherlands, 1957).
[9] J.P. Crutchfield and K. Young, Computation at the onset of chaos, in: W. Zurek, ed., *Complexity, Entropy and the Physics of Information* (Addison-Wesley, Reading, MA, 1989).

[10] J.J. Gibson, *The Ecological Approach to Visual Perception* (Houghton-Mifflin, Boston, MA).
[11] J. Gleick, *Chaos: Making a New Science* (Viking, New York, 1987).
[12] W.D. Hillis, Intelligence as emergent behavior; or, the songs of eden, *Daedelus* **117** (1988) 175–190.
[13] T.K. Landauer, How much do people remember? Some estimates on the quantity of learned information in long-term memory, *Cogn. Sci.* **10** (1986) 477–494.
[14] B. Mandelbrot, *The Fractal Geometry of Nature* (Freeman, San Francisco, CA, 1982).
[15] H. Moravec, *Mind Children* (Harvard University Press, Cambridge, MA, 1988).
[16] A. Newell, The knowledge level, *Artif. Intell.* **18** (1982) 87–127.
[17] A. Newell and H.A. Simon, Computer science as empirical inquiry: symbols and search, *Comm. ACM* **19** (3) (1976) 113–126.
[18] J.B. Pollack, Recursive distributed representation, *Artif. Intell.* **46** (1990) 77–105.
[19] J.B. Pollack, The induction of dynamical recognizers, *Mach. Learn.* **7** (1991) 227–252.
[20] R. Port and T. Van Gelder, Mind as motion (in preparation).
[21] D.E. Rumelhart, J.L. McClelland and the PDP Research Group, eds., *Parallel Distributed Processing: Experiments in the Microstructure of Cognition* (MIT Press, Cambridge, MA, 1986).
[22] M. Schroeder, *Fractals, Chaos, Power Laws* (Freeman, New York, 1991).
[23] P. Smolensky, Information processing in dynamical systems: foundations of harmony theory, in: D.E. Rumelhart, J.L. McClelland and the PDP Research Group, eds., *Parallel Distributed Processing: Experiments in the Microstructure of Cognition*, Vol. 1 (MIT Press, Cambridge, MA, 1986) 194–281.
[24] J. Von Neumann, *The Computer and the Brain* (Yale University Press, New Haven, CT, 1958).
[25] O. Wright, *How we Invented the Airplane* (Dover, New York, 1988).

An Evolutionary Algorithm that Constructs Recurrent Neural Networks

Peter J. Angeline, Gregory M. Saunders and Jordan B. Pollack
Laboratory for Artificial Intelligence Research
Computer and Information Science Department
The Ohio State University
Columbus, Ohio 43210
pja@cis.ohio-state.edu
saunders@cis.ohio-state.edu
pollack@cis.ohio-state.edu

Abstract

Standard methods for inducing both the structure and weight values of recurrent neural networks fit an assumed class of architectures to every task. This simplification is necessary because the interactions between network structure and function are not well understood. Evolutionary computation, which includes genetic algorithms and evolutionary programming, is a population-based search method that has shown promise in such complex tasks. This paper argues that genetic algorithms are inappropriate for network acquisition and describes an evolutionary program, called GNARL, that simultaneously acquires both the structure and weights for recurrent networks. This algorithm's empirical acquisition method allows for the emergence of complex behaviors and topologies that are potentially excluded by the artificial architectural constraints imposed in standard network induction methods.

To Appear in:

IEEE Transactions on Neural Networks

An Evolutionary Algorithm that Constructs Recurrent Neural Networks

Peter J. Angeline, Gregory M. Saunders and Jordan B. Pollack
Laboratory for Artificial Intelligence Research
Computer and Information Science Department
The Ohio State University
Columbus, Ohio 43210
pja@cis.ohio-state.edu
saunders@cis.ohio-state.edu
pollack@cis.ohio-state.edu

Abstract

Standard methods for inducing both the structure and weight values of recurrent neural networks fit an assumed class of architectures to every task. This simplification is necessary because the interactions between network structure and function are not well understood. Evolutionary computation, which includes genetic algorithms and evolutionary programming, is a population-based search method that has shown promise in such complex tasks. This paper argues that genetic algorithms are inappropriate for network acquisition and describes an evolutionary program, called GNARL, that simultaneously acquires both the structure and weights for recurrent networks. This algorithm's empirical acquisition method allows for the emergence of complex behaviors and topologies that are potentially excluded by the artificial architectural constraints imposed in standard network induction methods.

1.0 Introduction

In its complete form, network induction entails both *parametric* and *structural* learning [1], i.e., learning both weight values and an appropriate topology of nodes and links. Current methods to solve this task fall into two broad categories. *Constructive* algorithms initially assume a simple network and add nodes and links as warranted [2-8], while *destructive* methods start with a large network and prune off superfluous components [9-12]. Though these algorithms address the problem of topology acquisition, they do so in a highly constrained manner. Because they monotonically modify network structure, constructive and destructive methods limit the traversal of the available architectures in that once an architecture has been explored and determined to be insufficient, a new architecture is adopted, and the old becomes topologically unreachable. Also, these methods often use only a single predefined structural modification, such as "add a fully connected hidden unit," to generate successive topologies. This is a form of *structural hill climbing*, which is susceptible to becoming trapped at structural local minima. In addition, constructive and destructive algorithms make simplifying architectural assumptions to facilitate network induction. For example, Ash [2] allows only feedforward networks; Fahlman [6] assumes a restricted form of recurrence, and Chen et al. [7] explore only fully connected topologies. This creates a situation in which the task is forced into the architecture rather than the architecture being fit to the task.

These deficiencies of constructive and destructive methods stem from inadequate methods for assigning credit to structural components of a network. As a result, the heuristics used are overly-constrained to increase the likelihood of finding any topology to solve the problem. Ideally, the constraints for such a solution should come from the task rather than be implicit in the algorithm.

This paper presents GNARL, a network induction algorithm that simultaneously acquires both network topology and weight values while making minimal architectural restrictions and avoiding structural hill climbing. The algorithm, described in section 3, is an instance of evolutionary programming [13, 14], a class of evolutionary computation that has been shown to perform well at function optimization. Section 2 argues that this class of evolutionary computation is better suited for evolving neural networks than genetic algorithms [15, 16], a more popular class of evolutionary computation. Finally, section 4 demonstrates GNARL's ability to create recurrent networks for a variety of problems of interest.

2.0 Evolving Connectionist Networks

Evolutionary computation provides a promising collection of algorithms for structural and parametric learning of recurrent networks [17]. These algorithms are distinguished by their reliance on a *population* of search space positions, rather than a single position, to locate extrema of a function defined over the search space. During one search cycle, or *generation*, the members of the population are ranked according to a *fitness function*, and those with higher fitness are probabilistically selected to become *parents* in the next generation. New population members, called *offspring*, are created using specialized *reproduction heuristics*. Using the population, reproduction heuristics, and fitness function, evolutionary computation implements a nonmonotonic search that performs well in complex multimodal environments. Classes of evolutionary computation can be distinguished by examining the specific reproduction heuristics employed.

Genetic algorithms (GAs) [15, 16] are a popular form of evolutionary computation that rely chiefly on the reproduction heuristic of *crossover*.[1] This operator forms offspring by recombining representational components from two members of the population without regard to content. This purely structural approach to creating novel population members assumes that components of all parent representations may be freely exchanged without inhibiting the search process.

Various combinations of GAs and connectionist networks have been investigated. Much research concentrates on the acquisition of parameters for a fixed network architecture (e.g., [18 - 21]). Other work allows a variable topology, but disassociates structure acquisition from acquisition of weight values by interweaving a GA search for network topology with a traditional parametric training algorithm (e.g., backpropagation) over weights (e.g., [22, 23]). Some studies attempt to coevolve both the topology and weight values within the GA framework, but as in the connectionist systems described above, the network architectures are restricted (e.g., [24 - 26]). In spite of this collection of studies, current theory from both genetic algorithms and connectionism suggests that GAs are not well-suited for evolving networks. In the following section, the reasons for this mismatch are explored.

1. Genetic algorithms also employ other operators to manipulate the population, including a form of mutation, but their distinguishing feature is a heavy reliance on crossover.

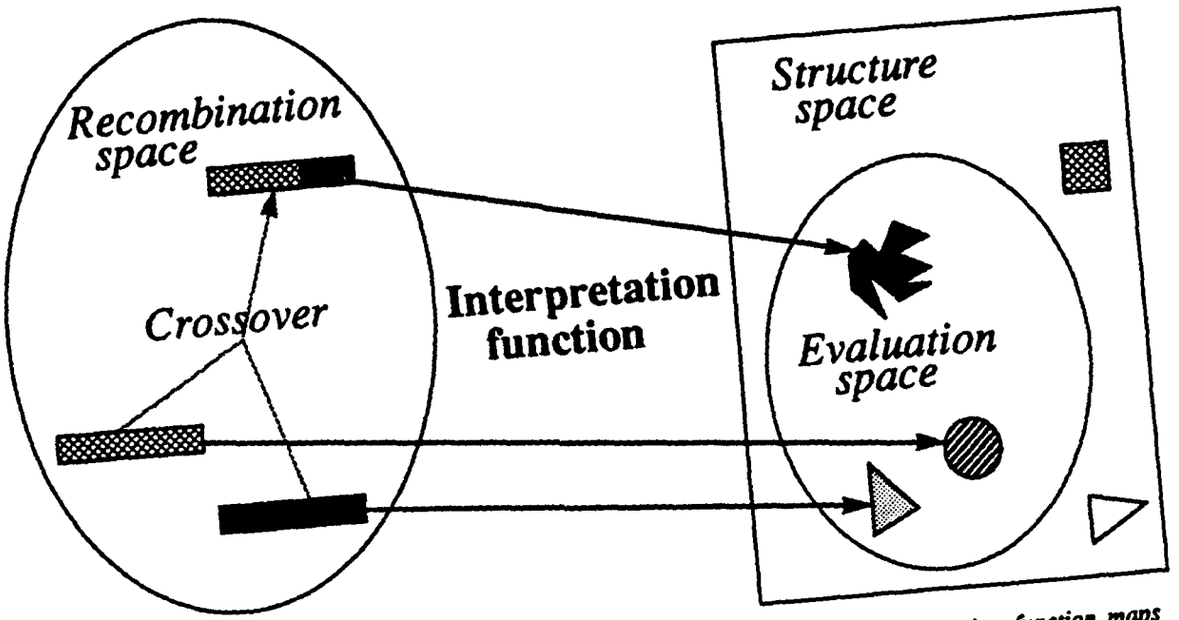


Figure 1. The dual representation scheme used in genetic algorithms. The interpretation function maps between the elements in recombination space on which the search is performed and the subset of structures that can be evaluated as potential task solutions.

2.1 Evolving Networks with Genetic Algorithms

Genetic algorithms create new individuals by recombining the representational components of two member of the population. Because of this commitment to structural recombination, GAs typically rely on two distinct representational spaces (Figure 1). *Recombination space*, usually defined over a set of fixed-length binary strings, is the set of structures to which the genetic operators are applied. It is here that the search actually occurs. *Evaluation space*, typically involving a problem-dependent representation, is the set of structures whose ability to perform a task is evaluated. In the case of using GAs to evolve networks, evaluation space is comprised of a set of networks. An *interpretation function* maps between these two representational spaces. Any set of finite-length bit strings cannot represent all possible networks, thus the evaluation space is restricted to a predetermined set of networks. By design, the dual representation scheme allows the GA to crossover the bit strings without any knowledge of their interpretation as networks. The implicit assumption is that the interpretation function is defined so that the bit strings created by the dynamics of the GA will map to successively better networks.

The dual representation of GAs is an important feature for searching in certain environments. For instance, when it is unclear how to search the evaluation space directly, and when there exists an interpretation function such that searching the space of bit strings by crossover leads to good points in evaluation space, then the dual representation is ideal. It is unclear, however, that there exists an interpretation function that makes dual representation beneficial for evolving neural networks. Clearly, the choice of interpretation function introduces a strong bias into the search, typically by excluding many potentially interesting and useful networks (another example of forcing

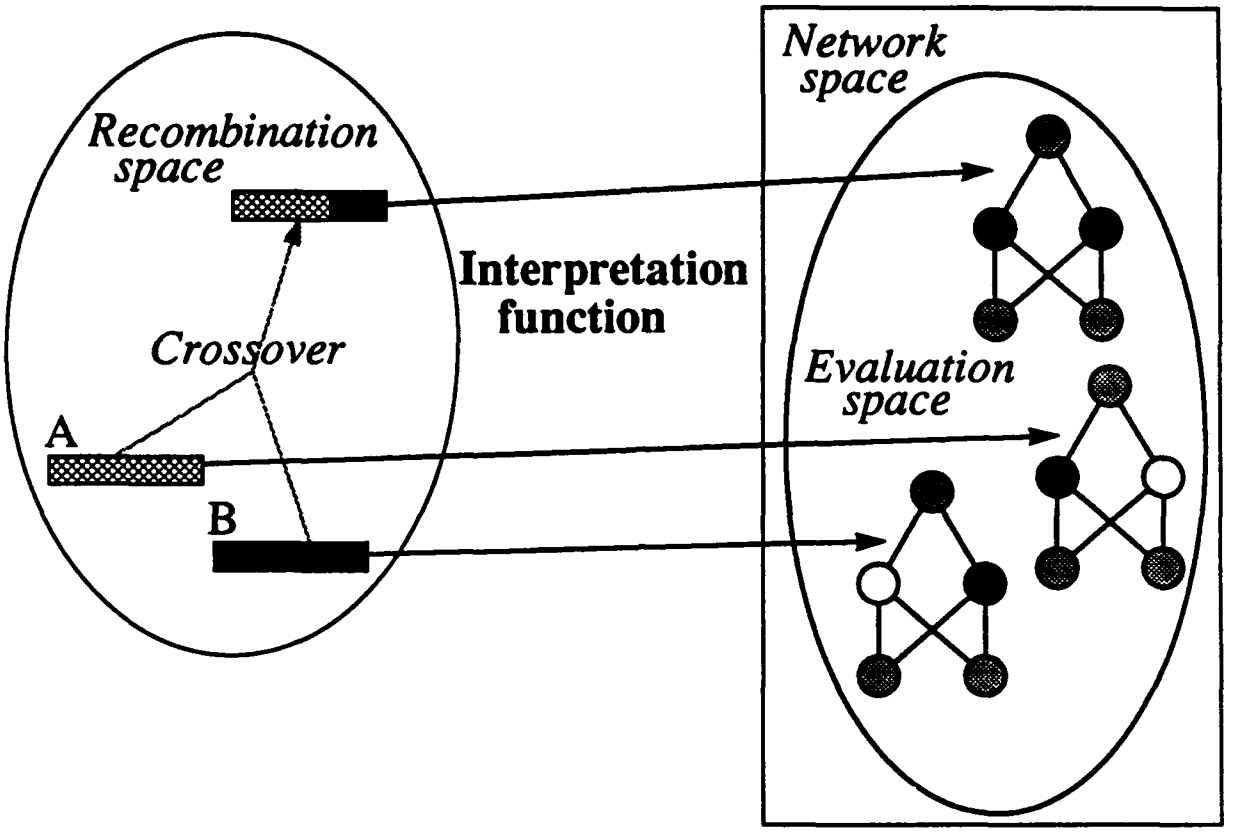


Figure 2. The competing conventions problem [29]. Bit strings A and B map to structurally and computationally equivalent networks that assign the hidden units in different orders. Because the bit strings are distinct, crossover is likely to produce an offspring that contains multiple copies of the same hidden node, yielding a network with less computational ability than either parent.

the task into an architecture). Moreover, the benefits of having a dual representation hinge on crossover being an appropriate evolutionary operator for the task for some particular interpretation function; otherwise, the need to translate between dual representations is an unnecessary complication.

Characterizing tasks for which crossover is a beneficial operator is an open question. Current theory suggests that crossover will tend to recombine short, connected substrings of the bit string representation that correspond to above-average task solutions when evaluated [16, 15]. These substrings are called *building blocks*, making explicit the intuition that larger structures with high fitness are built out of smaller structures with moderate fitness. Crossover tends to be most effective in environments where the fitness of a member of the population is reasonably correlated with the expected ability of its representational components [27]. Environments where this is not true are called *deceptive* [28].

There are three forms of deception when using crossover to evolve connectionist networks. The first involves networks that share both a common topology and common weights. Because the interpretation function may be many-to-one, two such networks need not have the same bit string representation (see Figure 2). Crossover will then tend to create offspring that contain repeated components, and lose the computational ability of some of the parents' hidden units. The

resulting networks will tend to perform worse than their parents because they do not possess key computational components for the task. Schaffer et al. [29] term this the *competing conventions* problem, and point out that the number of competing conventions grows exponentially with the number of hidden units.

The second form of deception involves two networks with identical topologies but different weights. It is well known that for a given task, a single connectionist topology affords multiple solutions for a task, each implemented by a unique *distributed representation* spread across the hidden units [30, 31]. While the removal of a small number of nodes has been shown to effect only minor alterations in the performance of a trained network [30, 31], the computational role each node plays in the overall representation of the task solution is determined purely by the presence and strengths of its interconnections. Furthermore, there need be no correlation between distinct distributed representations over a particular network architecture for a given task. This seriously reduces the chance that an arbitrary crossover operation between distinct distributed representations will construct viable offspring *regardless of the interpretation function used.*

Finally, deception can occur when the parents differ topologically. The types of distributed representations that can develop in a network vary widely with the number of hidden units and the network's connectivity. Thus, the distributed representations of topologically distinct networks have a greater chance of being incompatible parents. This further reduces the likelihood that crossover will produce good offspring.

In short, for crossover to be a viable operator when evolving networks, the interpretation function must somehow compensate for all the types of deceptiveness described above. This suggests that the complexity of an appropriate interpretation function will more than rival the complexity of the original learning problem. Thus, the prospect of evolving connectionist networks with crossover appears limited in general, and better results should be expected with reproduction heuristics that respect the uniqueness of the distributed representations. This point has been tacitly validated in the genetic algorithm literature by a trend towards a reduced reliance on binary representations when evolving networks (e.g. [32, 33]). Crossover, however, is still commonplace.

2.2 Networks and Evolutionary Programming

Unlike genetic algorithms, evolutionary programming (EP) [14,34] defines representation-dependent mutation operators that create offspring within a specific locus of the parent (see Figure 3). EP's commitment to mutation as the sole reproductive operator for searching over a space is preferable when there is no sufficient calculus to guide recombination by crossover, or when separating the search and evaluation spaces does not afford an advantage.

Relatively few previous EP systems have addressed the problem of evolving connectionist networks. Fogel et al. [35] investigate training feedforward networks on some classic connectionist problems. McDonnell and Waagen [36] use EP to evolve the connectivity of feedforward networks with a constant number of hidden units by evolving both a weight matrix and a connectivity matrix. Fogel [14], [37] uses EP to induce three-layer fully-connected feedforward networks with a variable number of hidden units that employ good strategies for playing Tic-Tac-Toe.

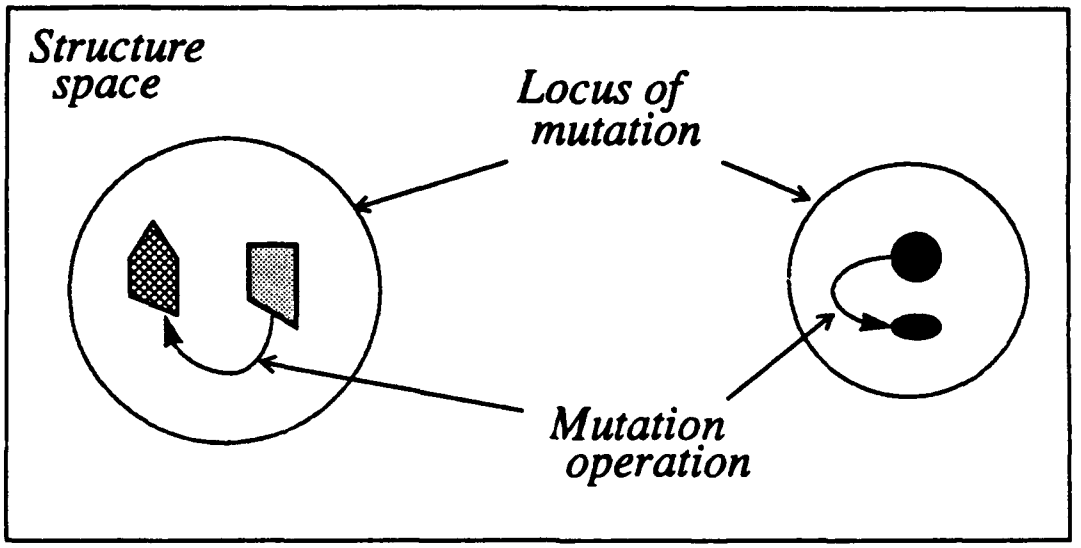


Figure 3. The evolutionary programming approach to modeling evolution. Unlike genetic algorithms, evolutionary programs perform search in the space of networks. Offspring created by mutation remain within a locus of similarity to their parents.

In each of the above studies, the mutation operator alters the parameters of network η by the function:

$$w = w + N(0, \alpha\varepsilon(\eta)) \qquad \forall w \in \eta \qquad \text{(EQ 1)}$$

where w is a weight, $\varepsilon(\eta)$ is the error of the network on the task (typically the mean squared error), α is a user-defined proportionality constant, and $N(\mu, \sigma^2)$ is a gaussian variable with mean μ and variance σ^2. The implementations of structural mutations in these studies differ somewhat. McDonnell and Waagen [36] randomly select a set of weights and alters their values with a probability based on the variance of the incident nodes' activation over the training set; connections from nodes with a high variance having less of a chance of being altered. The structural mutation used in [14, 37] adds or deletes a single hidden unit with equal probability

Evolutionary programming provides distinct advantages over genetic algorithms when evolving networks. First, EP manipulates networks directly, thus obviating the need for a dual representation and the associated interpretation function. Second, by avoiding crossover between networks in creating offspring, the individuality of each network's distributed representation is respected. For these reasons, evolutionary programming provides a more appropriate framework for simultaneous structural and parametric learning in recurrent networks. The GNARL algorithm, presented in the next section and investigated in the remainder of this paper, describes one such approach.

3.0 The GNARL Algorithm

GNARL, which stands for *GeNeralized Acquisition of Recurrent Links*, is an evolutionary algorithm that nonmonotonically constructs recurrent networks to solve a given task. The name

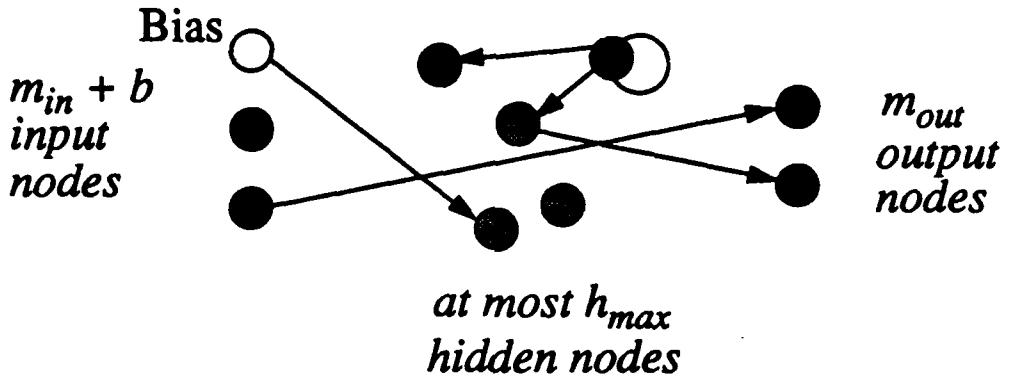


Figure 4. *Sample initial network. The number of input nodes (m_{in}) and number of output nodes (m_{out}) is fixed for a given task. The presence of a bias node (b = 0 or 1) as well as the maximum number of hidden units (h_{max}) is set by the user. The initial connectivity is chosen randomly (see text). The disconnected hidden node does not affect this particular network's computation, but is available as a resource for structural mutations.*

GNARL reflects the types of networks that arise from a generalized network induction algorithm performing both structural and parametric learning. Instead of having uniform or symmetric topologies, the resulting networks have "gnarled" interconnections of hidden units which more accurately reflect constraints inherent in the task.

The general architecture of a GNARL network is straightforward. The input and output nodes are considered to be provided by the task and are immutable by the algorithm; thus each network for a given task always has m_{in} input nodes and m_{out} output nodes. The number of hidden nodes varies from 0 to a user-supplied maximum h_{max}. Bias is optional; if provided in an experiment, it is implemented as an additional input node with constant value one. All non-input nodes employ the standard sigmoid activation function. Links use real-valued weights, and must obey three restrictions:

R_1: There can be no links *to* an input node.
R_2: There can be no links *from* an output node.
R_3: Given two nodes x and y, there is at most one link from x to y.

Thus GNARL networks may have no connections, sparse connections, or full connectivity. Consequently, GNARL's search space is:

S = {η: η is a network with real-valued weights,
η satisfies R_1-R_3,
η has m_{in} + b input nodes, where b=1 if a bias node is provided, and 0 otherwise,
η has m_{out} output nodes,
η has i hidden nodes, $0 \le i \le h_{max}$}

R_1-R_3 are strictly implementational constraints. Nothing in the algorithm described below hinges on S being pruned by these restrictions.

3.1 Selection, Reproduction and Mutation of Networks

GNARL initializes the population with randomly generated networks (see Figure 4). The number of hidden nodes for each network is chosen from a uniform distribution over a user-sup-

plied range. The number of initial links is chosen similarly from a second user-supplied range. The incident nodes for each link are chosen in accordance with the structural mutations described below. Once a topology has been chosen, all links are assigned random weights, selected uniformly from the range [-1, 1]. There is nothing in this initialization procedure that forces a node to have *any* incident links, let alone for a path to exist between the input and output nodes. In the experiments below, the number of hidden units for a network in the initial population was selected uniformly between one and five and the number of initial links varied uniformly between one and 10.

In each generation of search, the networks are first evaluated by a user-supplied fitness function f: $S \rightarrow \boldsymbol{R}$, where $\boldsymbol{R}$ represents the reals. Networks scoring in the top 50% are designated as the *parents* of the next generation; all other networks are discarded. This selection method is used in many EP algorithms although competitive methods of selection have also been investigated [14].

Generating an offspring involves three steps: copying the parent, determining the severity of the mutations to be performed, and finally mutating the copy. Network mutations are separated into two classes, corresponding with the types of learning discussed in [1]. *Parametric mutations* alter the value of parameters (link weights) currently in the network, whereas *structural mutations* alter the number of hidden nodes and the presence of links in the network, thus altering the space of parameters.

3.1.1 Severity of Mutations

The severity of a mutation to a given parent, η, is dictated by that network's temperature, $T(\eta)$:

$$T(\eta) = 1 - \frac{f(\eta)}{f_{\max}} \qquad \text{(EQ 2)}$$

where f_{max} is the maximum fitness for a given task. Thus, the temperature of a network is determined by how close the network is to being a solution for the task. This measure of the network's performance is used to anneal the structural and parametric similarity between parent and offspring, so that networks with a high temperature are mutated severely, and those with a low temperature are mutated only slightly (cf. [38]). This allows a coarse-grained search initially, and a progressively finer-grained search as a network approaches a solution to the task, a process described more concretely below.

3.1.2 Parametric Mutation of Networks

Parametric mutations are accomplished by perturbing each weight w of a network η with gaussian noise, a method motivated by [37, 14]. In that body of work, weights are modified as follows:

$$w = w + N(0, \alpha T(\eta)) \qquad \forall w \in \eta \qquad \text{(EQ 3)}$$

where α is a user-defined proportionality constant, and $N(\mu, \sigma^2)$ is a gaussian random variable as before. While large parametric mutations are occasionally necessary to avoid parametric local minima during search, it is more likely they will adversely affect the offspring's ability to perform better than its parent. To compensate, GNARL updates weights using a variant of equation 3. First, the *instantaneous temperature* $\hat{T}$ of the network is computed:

$$\hat{T}(\eta) = U(0, 1)\, T(\eta) \qquad \text{(EQ 4)}$$

where $U(0, 1)$ is a uniform random variable over the interval [0, 1]. This new temperature, varying from 0 to $T(\eta)$, is then substituted into equation 3:

$$w = w + N(0, \alpha\hat{T}(\eta)) \qquad \forall w \in \eta \qquad \text{(EQ 5)}$$

In essence, this modification lessens the frequency of large parametric mutations without disallowing them completely. In the experiments described below, α is one.

3.1.3 Structural Mutation of Networks

The structural mutations used by GNARL alter the number of hidden nodes and the connectivity between all nodes, subject to restrictions R_1-R_3 discussed earlier. To avoid radical jumps in fitness from parent to offspring, structural mutations attempt to preserve the behavior of a network. For instance, new links are initialized with zero weight, leaving the behavior of the modified network unchanged. Similarly, hidden units are added to the network without any incident connections. Links must be added by future structural mutations to determine how to incorporate the new computational unit. Unfortunately, achieving this behavioral continuity between parent and child is not so simple when removing a hidden node or link. Consequently, the deletion of a node involves the complete removal of the node and all incident links with no further modification to compensate for the behavioral change. Similarly, deleting a link removes that parameter from the network.

The selection of which node to remove is uniform over the collection of hidden nodes. Addition or deletion of a link is slightly more complicated in that a parameter identifies the likelihood that the link will originate from an input node or terminate at an output node. Once the class of incident node is determined, an actual node is chosen uniformly from the class. Biasing the link selection process in this way is necessary when there is a large differential between the number of hidden nodes and the number of input or output nodes. This parameter was set to 0.2 in the experiments described in the next section.

Research in [14] and [37] uses the heuristic of adding or deleting at most a single fully connected node per structural mutation. Therefore, it is possible for this method is to become trapped at a structural local minima, although this is less probable than in nonevolutionary algorithms given that several topologies may be present in the population. In order to more effectively search the range of network architectures, GNARL uses a severity of mutation for each separate structural mutation. A unique user-defined interval specifying a range of modification is associated with each of the four structural mutations. Given an interval of $[\Delta_{min}, \Delta_{max}]$ for a particular structural mutation, the number of modifications of this type made to an offspring is given by:

$$\Delta_{min} + \lfloor U[0, 1]\, \hat{T}(\eta)\, (\Delta_{max} - \Delta_{min}) \rfloor \qquad \text{(EQ 6)}$$

Thus the number of modifications varies uniformly over a shrinking interval based on the parent network's fitness. In the experiments below, the maximum number of nodes added or deleted was three while the maximum number of links added or deleted was five. The minimum number for each interval was always one.

3.2 Fitness of a Network

In evolving networks to perform a task, GNARL does not require an explicit target vector – all that is needed is the feedback given by the fitness function *f*. But if such a vector is present, as in supervised learning, there are many ways of transforming it into a measure of fitness. For example, given a training set $\{(x_1, y_1), (x_2, y_2), ...\}$, three possible measures of fitness for a network η are sum of square errors (equation 7), sum of absolute errors (equation 8), and sum of exponential absolute errors (equation 9):

$$\sum_i (y_i - Out(\eta, x_i))^2 \qquad \text{(EQ 7)}$$

$$\sum_i |y_i - Out(\eta, x_i)| \qquad \text{(EQ 8)}$$

$$\sum_i e^{|y_i - Out(\eta, x_i)|} \qquad \text{(EQ 9)}$$

Furthermore, because GNARL explores the space of networks by mutation and selection, the choice of fitness function does not alter the mechanics of the algorithm. To show GNARL's flexibility, each of these fitness functions will be demonstrated in the experiments below.

4.0 Experiments

In this section, GNARL is applied to several problems of interest. The goal in this section is to demonstrate the abilities of the algorithm on problems from language induction to search and collection. The various parameter values for the program are set as described above unless otherwise noted.

4.1 Williams' Trigger Problem

As an initial test, GNARL induced a solution for the *enable-trigger* task proposed in [39]. Consider the finite state generator shown in Figure 5. At each time step the system receives two input bits, (*a*, *b*), representing "enable" and "trigger" signals, respectively. This system begins in state S_1, and switches to state S_2 only when enabled by a=1. The system remains in S_2 until it is triggered by b=1, at which point it outputs 1 and resets the state to S_1. So, for instance, on an input stream {(0, 0), (0, 1), (1, 1), (0, 1)}, the system will output {0, 0, 0, 1} and end in S_1. This simple problem allows an indefinite amount of time to pass between the enable and the trigger inputs;

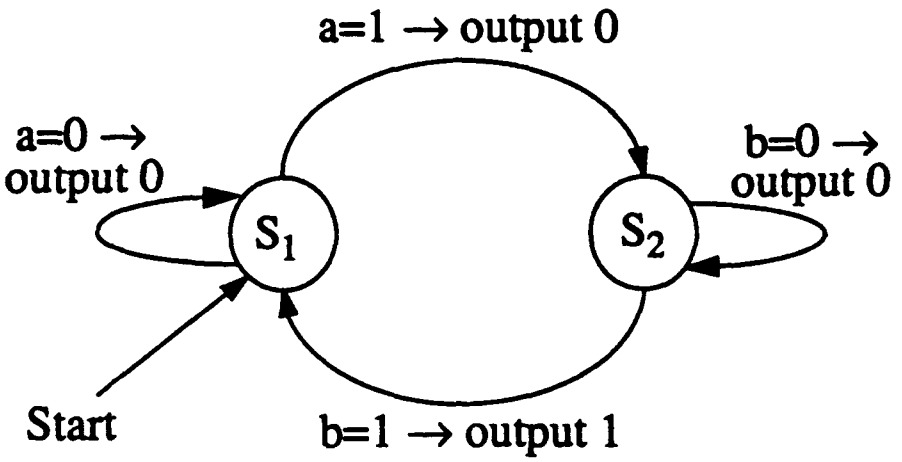


Figure 5. An FSA that defines the enable-trigger task [39]. The system is given a data stream of bit pairs $\{(a_1, b_1), (a_2, b_2), ...\}$, and produces an output of 0's and 1's. To capture this system's input/output behavior, a connectionist network must learn to store state indefinitely.

thus no finite length sample of the output stream will indicate the current state of the system. This forces GNARL to develop networks that can preserve state information indefinitely.

The fitness function used in this experiment was the sum of exponential absolute errors (equation 9). Population size was 50 networks with the maximum number of hidden units restricted to six. A bias node was provided in each network in this initial experiment, ensuring that an activation value of 1 was always available. Note that this does not imply that each node had a nonzero bias; links to the bias node had to be acquired by structural mutation.

Training began with all two input strings of length two, shown in Table1. After 118 generations (3000 network evaluations[2]), GNARL evolved a network which solved this task for the strings in Table 1 within tolerance of 0.3 on the output units. The training set was then increased to include all 64 input strings of length three and evolution of the networks was allowed to continue. After an additional 422 generations, GNARL once again found a suitable network. At this point, the difficulty of the task was increased a final time by training on all 256 strings of length four. After another 225 generations (~20000 network evaluations total) GNARL once again found a network to solve this task, shown in Figure 6b. Note that there are two completely isolated nodes. Given the fitness function used in this experiment, the two isolated nodes do not effect the network's viability. To investigate the generalization of this network, it was tested over all 4096 unique strings of length six. The outputs were rounded off to the nearest integer, testing only the network's separation of the strings. The network performed correctly on 99.5% of this novel set, generating incorrect responses for only 20 strings.

Figure 7 shows the connectivity of the population member with the best fitness for each generation over the course of the run. Initially, the best network is sparsely-connected and remains sparsely-connected throughout most of the run. At about generation 400, the size and connectivity

2. Number of networks evaluated = |population| + generations * |population| * 50% of the population removed each generation, giving 50 + 118 * 50 * 0.5 = 3000 network evaluations for this trial.

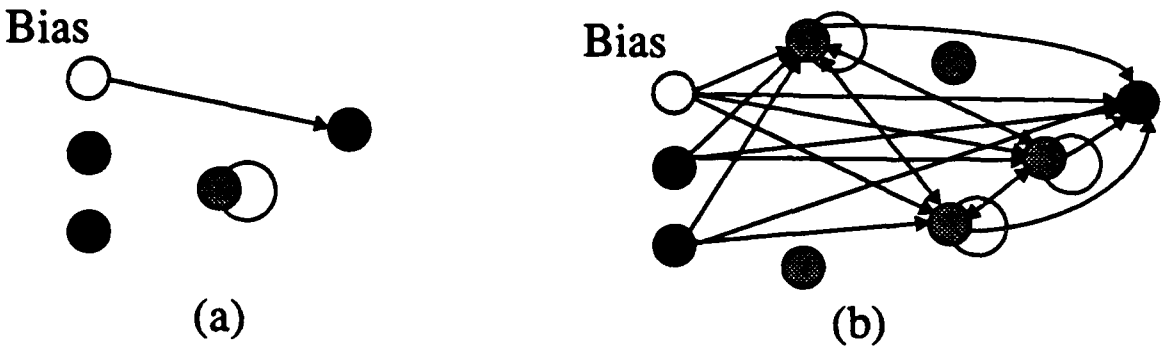


Figure 6. Connectivity of two recurrent networks found in the enable-trigger experiment. (a) The best network of generation 1. (b) The best network of generation 765. This network solves the task for all strings of length eight.

increases dramatically only to be overtaken by the relatively sparse architecture shown in Figure 6b on the final generation. Apparently, this more sparsely connected network evolved more quickly than the more full architectures that were best in earlier generations. The oscillations between different network architectures throughout the run reflects the development of such competing architectures in the population.

4.2 Inducing Regular Languages

A current topic of research in the connectionist community is the induction of finite state automata (FSAs) by networks with second-order recurrent connections. For instance, Pollack [40] trains sequential cascaded networks (SCNs) over a test set of languages, provided in [41] and

Input	Target Output
{(0, 0), (0, 0)}	{0, 0}
{(0, 0), (0, 1)}	{0, 0}
{(0, 0), (1, 0)}	{0, 0}
{(0, 0), (1, 1)}	{0, 0}
{(0, 1), (0, 0)}	{0, 0}
{(0, 1), (0, 1)}	{0, 0}
{(0, 1), (1, 0)}	{0, 0}
{(0, 1), (1, 1)}	{0, 0}

Input	Target Output
{(1, 0), (0, 0)}	{0, 0}
{(1, 0), (0, 1)}	{0, 1}
{(1, 0), (1, 0)}	{0, 0}
{(1, 0), (1, 1)}	{0, 1}
{(1, 1), (0, 0)}	{0, 0}
{(1, 1), (0, 1)}	{0, 1}
{(1, 1), (1, 0)}	{0, 0}
{(1, 1), (1, 1)}	{0, 1}

Table 1. Initial training data for enable-trigger task.

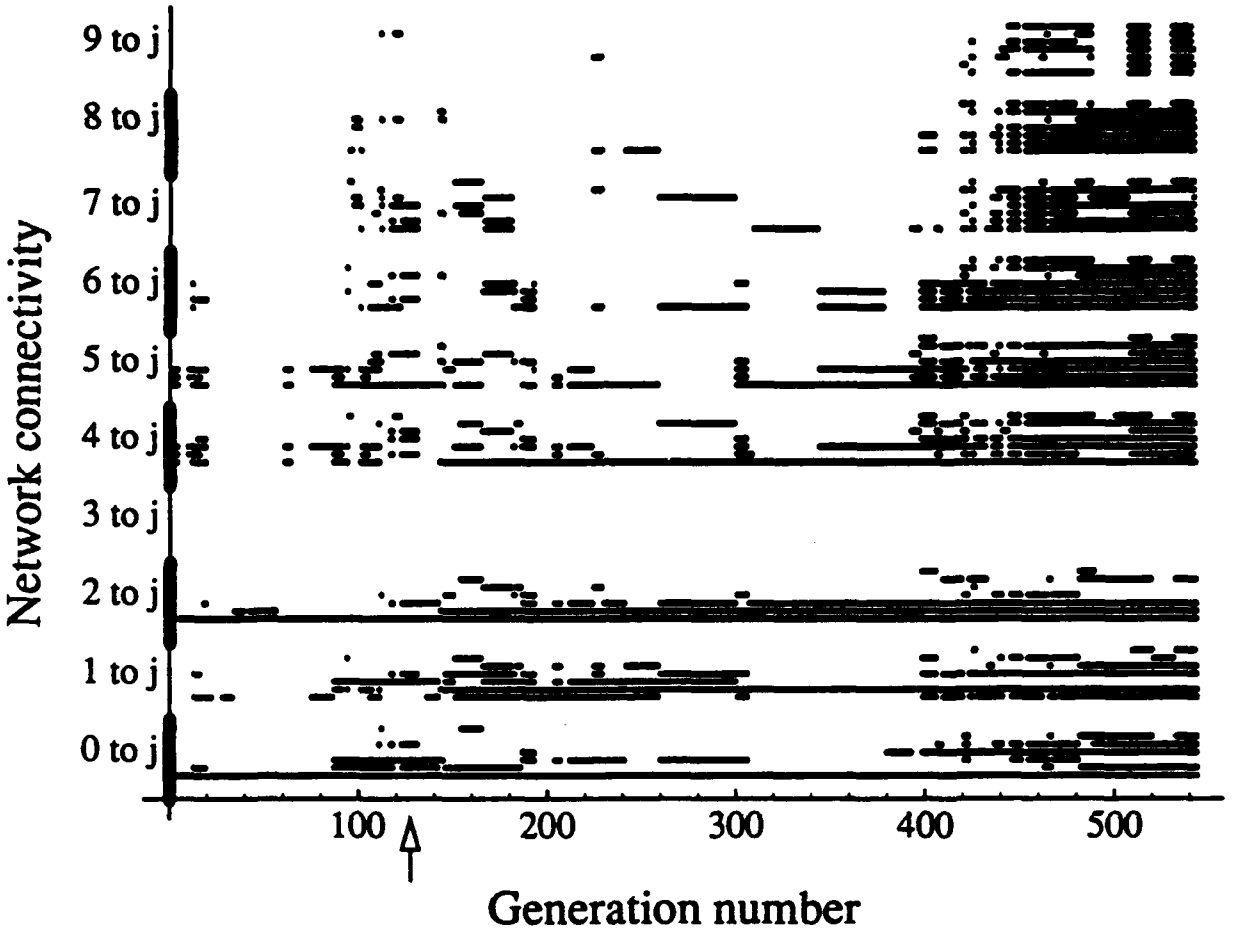


*Figure 7. Different network topologies explored by GNARL during the first 540 generations on the enable-trigger problem. The presence of a link between node i and j at generation g is indicated by a dot at position (g, 10 * i + j) in the graph. Note that because node 3 is the output node, there are no connections from it throughout the run. The arrow designates the point of transition between the first two training sets.*

shown in Table 2, using a variation of backpropagation. An interesting result of this work is that the number of states used by the network to implement finite state behavior is potentially infinite. Other studies using the training sets in [41] have investigated various network architectures and training methods, as well as algorithms for extracting FSAs from the trained architectures [42 - 45].

An explicit collection of positive and negative examples, shown in Table 3, that pose specific difficulties for inducing the intended languages is offered in [41]. Notice that the training sets are unbalanced, incomplete and vary widely in their ability to strictly define the intended regular language. GNARL's ability to learn and generalize from these training sets was compared against the training results reported for the second-order architecture used in [42]. Notice that all the languages in Table 2 require recurrent network connections in order to induce the language completely. The type of recurrence needed for each language varies widely. For instance, languages 1 through 4 require an incorrect input be remembered indefinitely, forcing the network to develop an analog version of a trap state. Networks for language 6, however, must parse and count indi-

Language	*Description*
1	1^*
2	$(1\ 0)^*$
3	no odd length 0 strings anytime after an odd length 1 string
4	no more than two 0s in a row
5	an even sum of 10s and 01s, pairwise
6	(number of 1s - number of 0s) mod 3 = 0
7	$0^*1^*0^*1^*$

Table 2. Regular languages to be induced.

vidual inputs, potentially changing state from accept to reject or vice versa on each successive input.

The results obtained in [42] are summarized in Table 4. The table shows the number of networks evaluated to learn the training set and the accuracy of generalization for the learned network to the intended regular language. Accuracy is measured as the percentage of strings of

Language	*Positive Instances*	*Negative Instances*
1	ε, 1, 11, 111, 1111, 11111, 111111, 1111111, 11111111	0, 10, 01, 00, 011, 110, 000, 11111110, 10111111
2	ε, 10, 1010, 101010, 10101010, 10101010101010	1, 0, 11, 00, 01, 101, 100, 1001010, 10110, 110101010
3	ε, 1, 0, 01, 11, 00, 100, 110, 111, 000, 100100, 110000011100001, 111101100010011100	10, 101, 010, 1010, 110, 1011, 10001, 111010, 1001000, 11111000, 0111001101, 11011100110
4	ε, 1, 0, 10, 01, 00, 100100, 001111110100, 0100100100, 11100, 010	000, 11000, 0001, 000000000, 00000, 0000, 11111000011, 1101010000010111, 1010010001
5	ε, 11, 00, 001, 0101, 1010, 1000111101, 1001100001111010, 111111, 0000	1, 0, 111, 010, 000000000, 1000, 01, 10, 1110010100, 0101111111110, 0001, 011
6	ε, 10, 01, 1100, 101010, 111, 000000, 0111101111, 100100100	1, 0, 11, 00, 101, 011, 11001, 1111, 00000000, 010111, 10111101111, 1001001001
7	ε, 1, 0, 10, 01, 11111, 000, 00110011, 0101, 0000100001111, 00100, 011111011111, 00	1010, 00110011000, 0101010101, 1011010, 10101, 010100, 101001, 100100110101

Table 3. Training sets for the languages of Table 2 from [41].

Language	*Average evaluations*	*Average % accuracy*	*Fewest evaluations*	*Best % accuracy*
1	3033.8	88.98	28	100.0
2	4522.6	91.18	807	100.0
3	12326.8	64.87	442	78.31
4	4393.2	42.50	60	60.92
5	1587.2	44.94	368	66.83
6	2137.6	23.19	306	46.21
7	29(0	36.97	373	55.74

Table 4. Speed and generalization results reported by [42] for learning the data sets of Table 3.

length 10 or less that are correctly classified by the network. For comparison, the table lists both the average and best performance of the five runs reported in [42].

This experiment used a population of 50 networks, each limited to at most eight hidden units. Each run lasted at most 1000 generations, allowing a maximum of 25050 networks to be evaluated for a single data set. Two experiments were run for each data set, one using the sum of absolute errors (SAE) and the other using sum of square errors (SSE). The error for a particular string was computed only for the final output of the network after the entire string plus three trailing "null" symbols had been entered, one input per time step. The concatenation of the trailing null symbols was used to identify the end of the string and allow input of the null string, a method also used in [42]. Each network had a single input and output and no bias node was provided. The three possible logical inputs for this task, 0, 1, and null, were represented by activations of -1, 1, and 0, respectively. The tolerance for the output value was 0.1, as in [42].

Table 5 shows fo. both fitness functions the number of evaluations until convergence and the accuracy of the best evolved network. Only four of the runs, each of those denoted by a '+' in the table, failed to produce a network with the specified tolerance in the allotted 1000 generations. In the runs using SAE, the two runs that did not converge had not separated a few elements of the associated training set and appeared to be far from discovering a network that could correctly classify the complete training set. Both of the uncompleted runs using SSE successfully separated the data sets but had not done so to the 0.1 tolerance within the 1000 generation limit. Figure 8 compares the number of evaluations by GNARL to the average number of evaluations reported in [42]. As the graph shows, GNARL consistently evaluates more networks, but not a disproportionate number. Considering that the space of networks being searched by GNARL is much larger than the space being searched by [42], these numbers appear to be within a tolerable increase.

The graph of Figure 9compares the accuracy of the GNARL networks to the *average* accuracy found in [42] over five runs. The GNARL networks consistently exceeded the average accuracy found in [42].

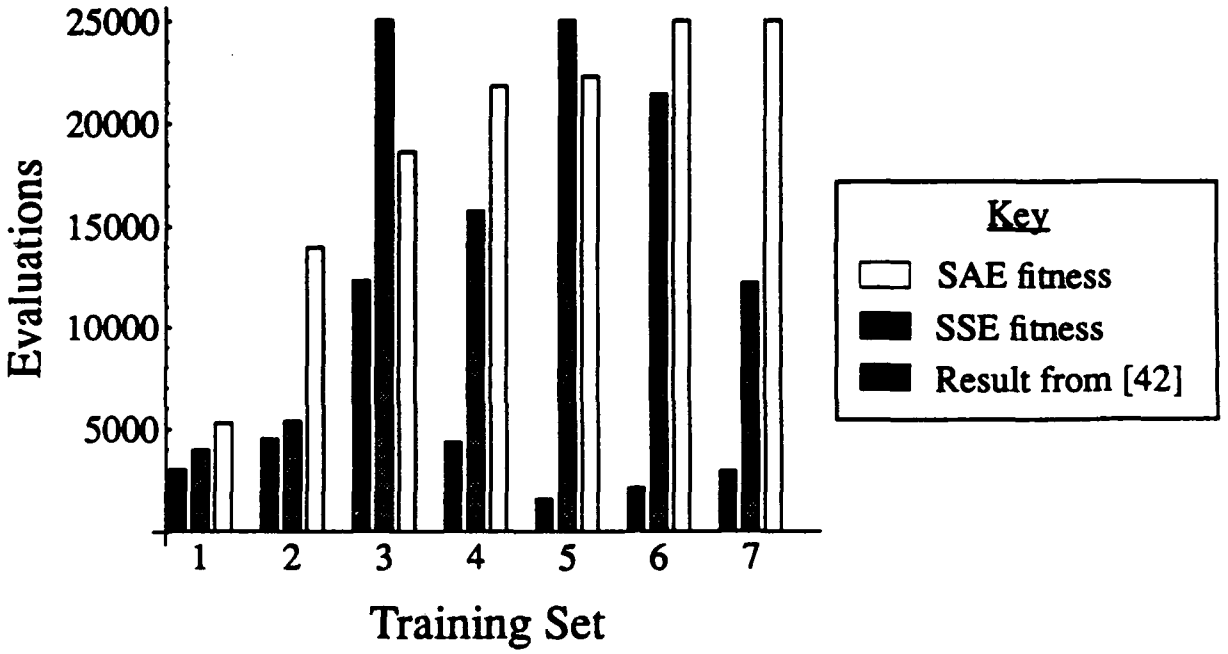


Figure 8. The number of network evaluations required to learn the seven data sets of Table 3. GNARL (using both SAE and SSE fitness measures) compared to the average number of evaluations for the five runs described in [42].

These results demonstrate GNARL's ability to simultaneously acquire the topology and weights of recurrent networks, and that this can be done within a comparable number of network evaluations as training a network with static architecture on the same task. GNARL also appears to generalize better consistently, possibly due to its selective inclusion and exclusion of some links.

Language	*Evaluations (SAE)*	*% Accuracy (SAE)*	*Evaluations (SSE)*	*% Accuracy (SSE)*
1	3975	100.00	5300	99.27
2	5400	96.34	13975	73.33
3	25050^{+}	58.87	18650	68.00
4	15775	92.57*	21850	57.15
5	25050^{+}	49.39	22325	51.25
6	21475	55.59*	25050^{+}	44.11
7	12200	71.37*	25050^{+}	31.46

Table 5. Speed and generalization results for GNARL to train recurrent networks to recognize the data sets of Table 3.

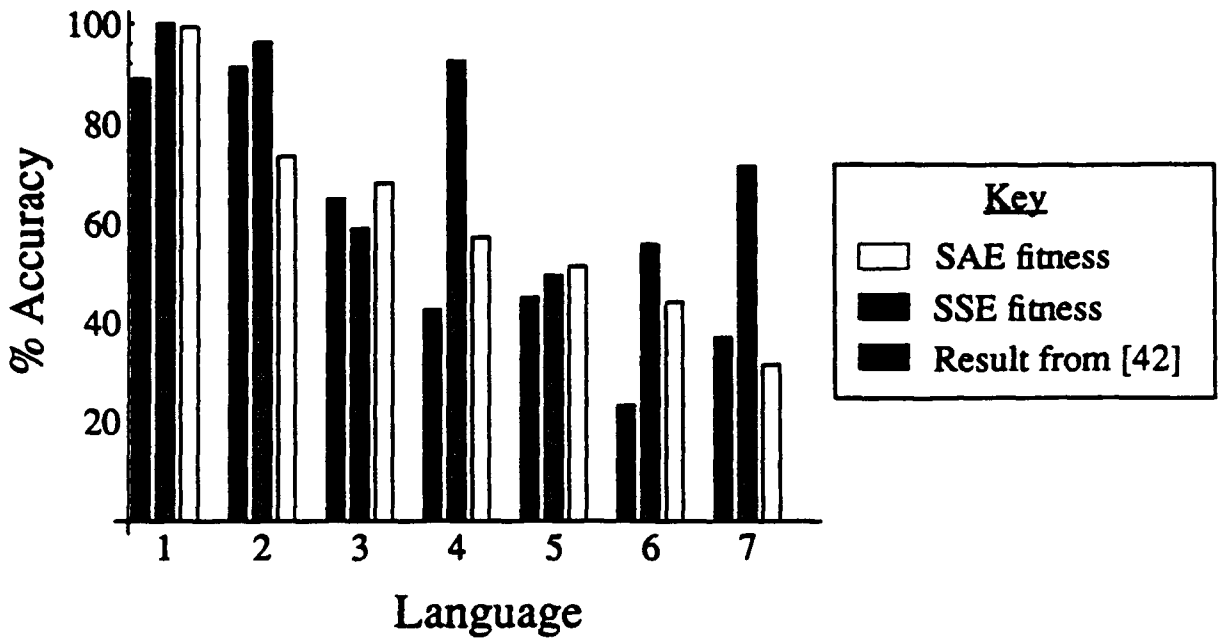


Figure 9. Percentage accuracy of evolved networks on languages in Table 2. GNARL (using SAE and SSE fitness measures) compared to average accuracy of the five runs in [42].

4.3 The Ant Problem

GNARL was tested on a complex search and collection task – the *Tracker* task described in [46], and further investigated in [47]. In this problem, a simulated ant is placed on a two-dimensional toroidal grid that contains a trail of food. The ant traverses the grid, collecting any food it contacts along the way. The goal of the task is to discover an ant which collects the maximum number of pieces of food in a given time period. (Figure 10).

Following [46], each ant is controlled by a network with two input nodes and four output nodes (Figure 11). The first input node denotes the presence of food in the square directly in front of the ant; the second denotes the absence of food in this same square, restricting the possible legal inputs to the network to (1, 0) or (0, 1). Each of the four output units corresponds to a unique action: move forward one step, turn left 90°, turn right 90°, or no-op. At each step, the action whose corresponding output node has maximum activation is performed. As in the original study [46], no-op allows the ant to remain at a fixed position while activation flows along recurrent connections. Fitness is defined as the number of grid positions cleared within 200 time steps. The task is difficult because simple networks can perform surprisingly well; the network shown in Figure 11 collects 42 pieces of food before spinning endlessly at position A (in Figure 10), illustrating a very high local maximum in the search space.

The experiment used a population of 100 networks, each limited to at most nine hidden units, and did not provide a bias node. In the first run (2090 generations), GNARL found a network (Figure 12b) that clears 81 grid positions within the 200 time steps. When this ant is run for an additional 119 time steps, it successfully clears the entire trail. To understand how the network traverses the path of food, consider the simple FSA shown in Figure 13, hand-crafted in [46] as an approximate solution to the problem. This simple machine receives a score of 81 in the allotted

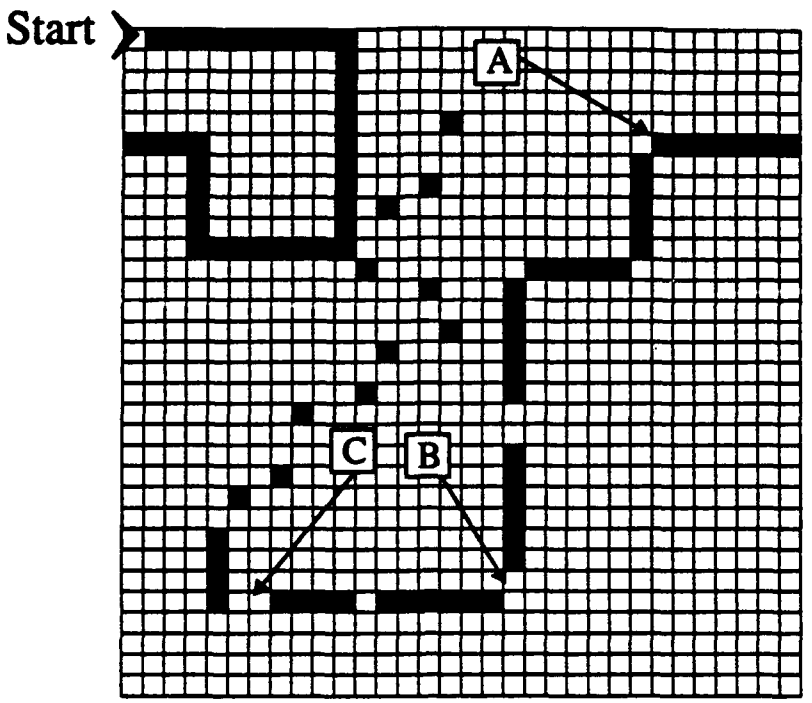


Figure 10. The ant problem. The trail is connected initially, but becomes progressively more difficult to follow. The underlying 2-d grid is toroidal, so that position "A" is the first break in the trail – it is simple to reach this point. Positions "B" and "C" indicate the only two positions along the trail where the ant discovered in run 1 behaves differently from the 5-state FSA of [46] (see Figure 13).

200 time steps, and clears the entire trail only five time steps faster than the network in Figure 12b. A step by step comparison indicates there is only a slight difference between the two. GNARL's evolved network follows the general strategy embodied by this FSA at all but two places, marked as positions B and C in Figure 10. Here the evolved network makes a few additional moves, accounting for the slightly longer completion time.

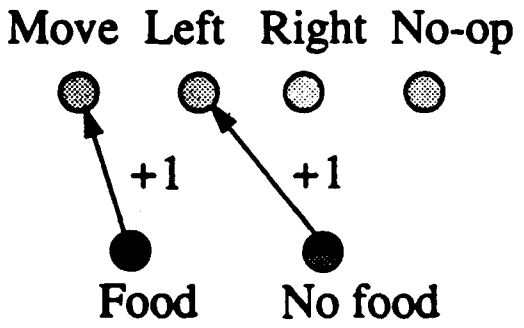
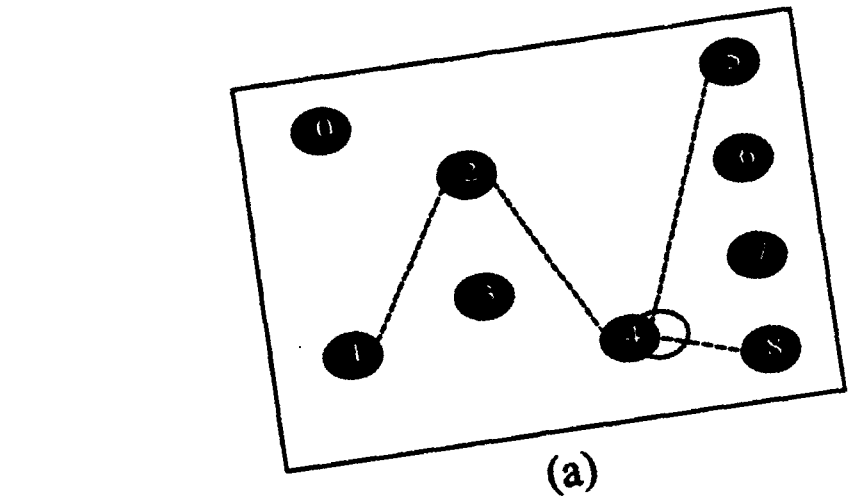
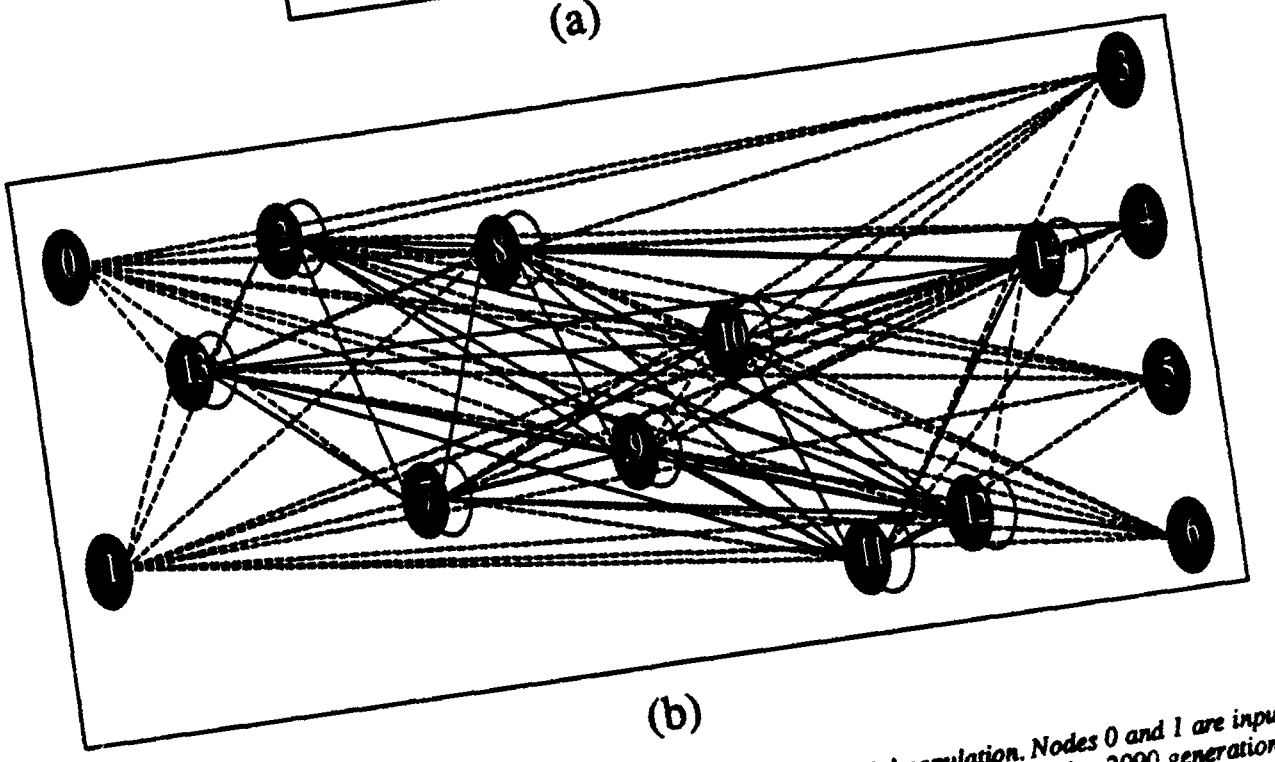


Figure 11. The semantics of the I/O units for the ant network. The first input node denotes the presence of food in the square directly in front of the ant; the second denotes the absence of food in this same square. This particular network finds 42 pieces of food before spinning endlessly in place at position P, illustrating a very deep local minimum in the search space.



(a)



(b)

Figure 12. The Tracker Task, first run. (a) The best network in the initial population. Nodes 0 and 1 are input, nodes 5-8 are output, and nodes 2-4 are hidden nodes. (b) Network induced by GNARL after 2090 generations. Forward links are dashed; bidirectional links and loops are solid. The light gray connection between nodes 8 and 13 is the sole backlink. This network clears the trail in 319 epochs.

Figure 14 illustrates the strategy the network uses to implement the FSA by showing the state of the output units of the network over three different sets. Each point is a triple of the form (*move*, *right*, *left*).[3] Figure 14a shows the result of supplying to the network 200 "food" inputs – a fixed point that executes "Move." Figure 14b shows the sequence of states reached when 200 "no food" signals are supplied to the network – a collection of points describing a limit cycle of length five that repeatedly executes the sequence "Right, Right, Right, Right, Move." These two attractors determine the response of the network to the task (Figure 14c, d); the additional points in Fig-

3. No-op is not shown because it was never used in the final network.

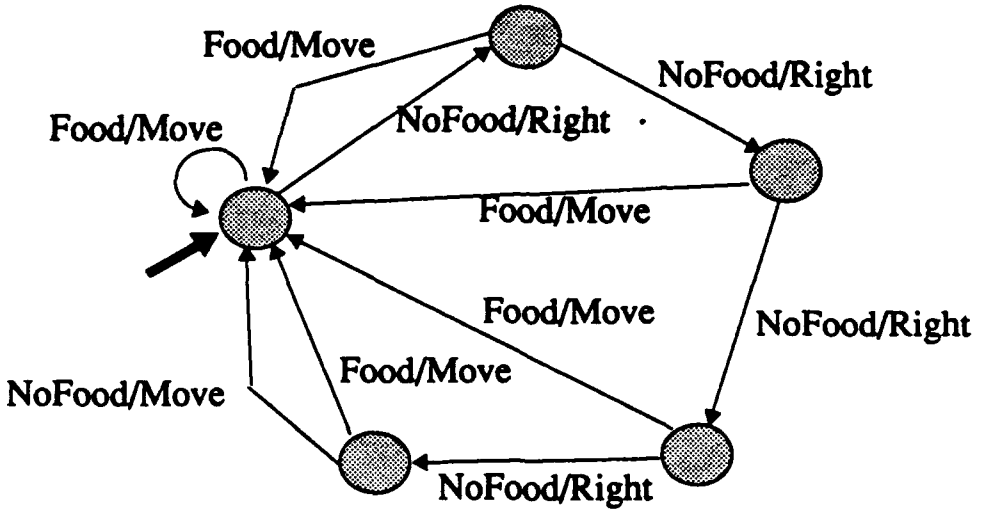


Figure 13. FSA hand-crafted for the Tracker task in [46]. The large arrow indicates the initial state. This simple system implements the strategy "move forward if there is food in front of you, otherwise turn right four times, looking for food. If food is found while turning, pursue it, otherwise, move forward one step and repeat." This FSA traverses the entire trail in 314 steps, and gets a score of 81 in the allotted 200 time steps.

ure 14c are transients encountered as the network alternates between these attractors. The differences in the number of steps required to clear the trail between the FSA of Figure 13 and GNARL's network arise due to the state of the hidden units when transferring from the "food" attractor to the "no food" attractor.

However, not all evolved network behaviors are so simple as to approximate an FSA [40]. In a second run (1595 generations) GNARL induced a network that cleared 82 grid points within the 200 time steps. Figure 15 demonstrates the behavior of this network. Once again, the "food" attractor, shown in Figure 15a, is a single point in the space that always executes "Move." The "no food" behavior, however, is not an FSA; instead, it is a quasiperiodic trajectory of points shaped like a "D" in output space (Figure 15b). The placement of the "D" is in the "Move / Right" corner of the space and encodes a complex alternation between these two operations (see Figure 15d).

In contrast, research in [46] uses a genetic algorithm on a population of 65,536 bit strings with a direct encoding to evolve only the weights of a neural network with five hidden units to solve this task. The particular network architecture in [46] uses Boolean threshold logic for the hidden units and an identity activation function for the output units. The first GNARL network was discovered after evaluating a total of 104,600 networks while the second was found after evaluating 79,850. The experiment reported in [46] discovered a comparable network after about 17 generations. Given [46] used a population size of 65,536 and replaced 95% of the population each generation, the total number of network evaluations to acquire the equivalent network was 1,123,942. This is 10.74 and 14.07 times the number of networks evaluated by GNARL in the two runs. In spite of the differences between the two studies, this significant reduction in the number of evaluations provides empirical evidence that crossover may not be best suited to the evolution of networks.

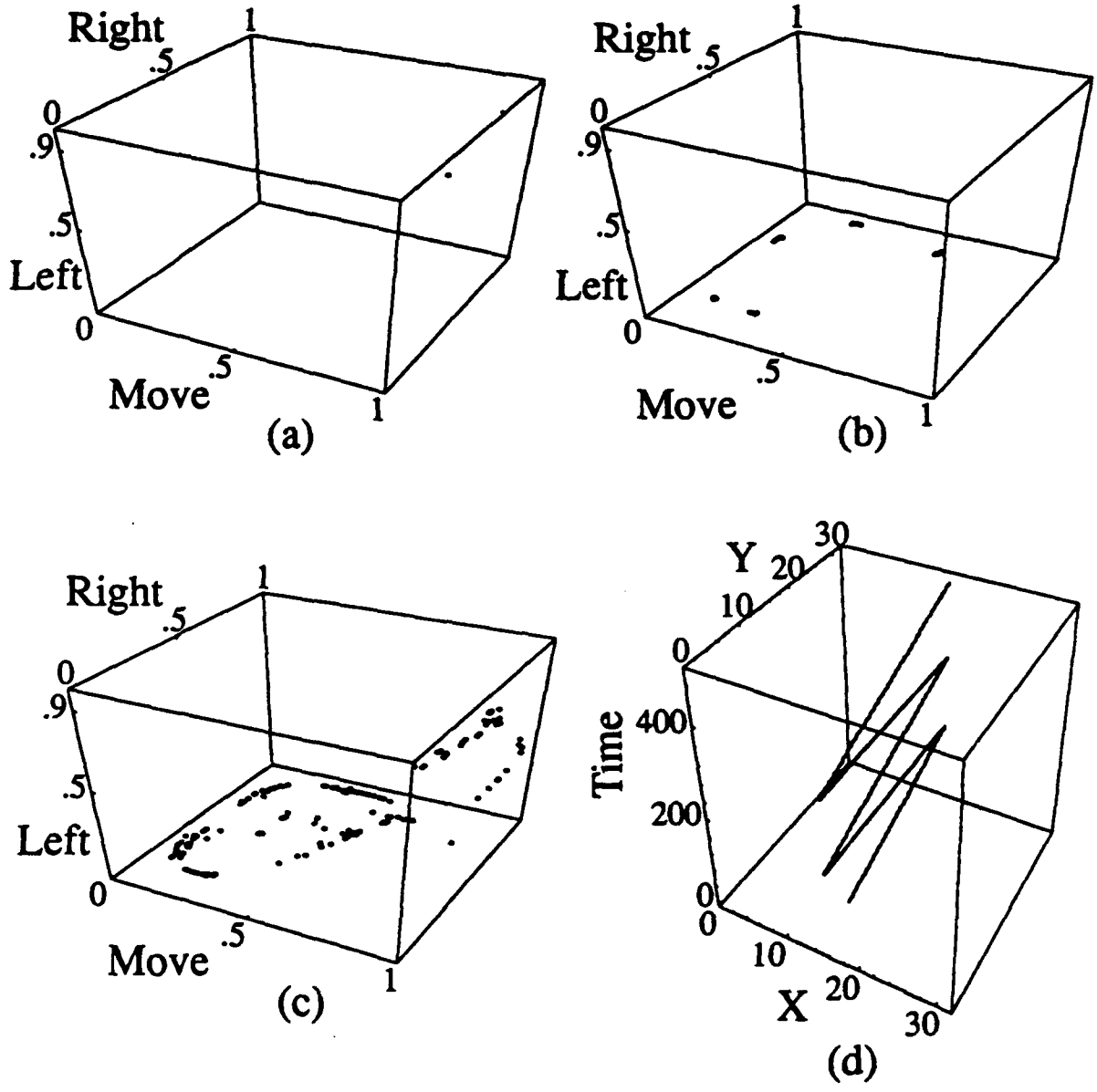


Figure 14. Limit behavior of the network that clears the trail in 319 steps. Graphs show the state of the output units Move, Right, Left. (a) Fixed point attractor that results for sequence of 500 "food" signals; (b) Limit cycle attractor that results when a sequence of 500 "no food" signals is given to network; (c) All states visited while traversing the trail; (d) The path of the ant on an empty grid. The Z axis represents time. Note that x is fixed, and y increases monotonically at a fixed rate. The large jumps in y position are artifacts of the toroidal grid.

5.0 Conclusions

Allowing the task to specify an appropriate architecture for its solution should, in principle, be the defining aspect of the complete network induction problem. By restricting the space of networks explored, constructive, destructive, and genetic algorithms only partially address the problem of topology acquisition. GNARL's architectural constraints R_1-R_3 similarly reduce the search space, but to a far less degree. Furthermore, none of these constraints is necessary, and their

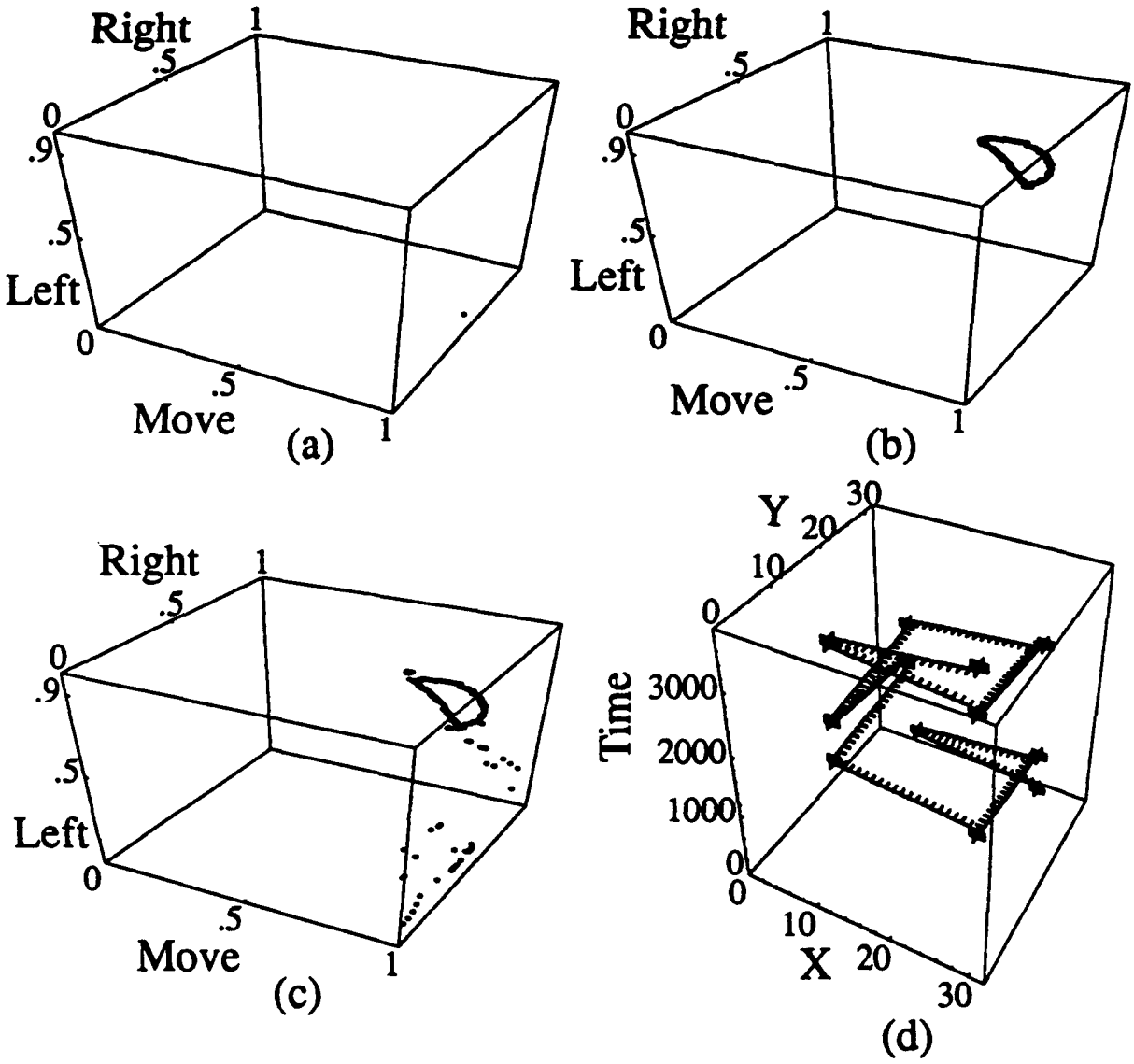


Figure 15. Limit behavior of the network of the second run. Graphs show the state of the output units Move, Right, Left. (a) Fixed point attractor that results for sequence of 3500 "food" signals; (b) Limit cycle attractor that results when a sequence of 3500 "no food" signals is given to network; (c) All states visited while traversing the trail; (d) The path of the ant on an empty grid. The z axis represents time. The ant's path is comprised of a set of "railroad tracks." Along each track, tick marks represent back and forth movement. At the junctures between tracks, a more complicated movement occurs. There are no artifacts of the toroidal grid in this plot, all are actual movements (cf. Figure 14d).

removal would affect only ease of implementation. In fact, no assumed features of GNARL's networks are essential for the algorithm's operation. GNARL could even use nondifferentiable activation functions, a constraint for backpropagation.

GNARL's minimal representational constraints would be meaningless if not complemented by appropriate search dynamics to traverse the space of networks. First, unlike constructive and destructive algorithms, GNARL permits a nonmonotonic search over the space of network topol-

ogies. Consider that in monotonic search algorithms, the questions of *when* and *how* to modify structure take on great significance because a premature topological change cannot be undone. In contrast, GNARL can revisit a particular architecture at any point, but for the architecture to be propagated it must confer an advantage over other competing topologies. Such a non-linear traversal of the space is imperative for acquiring appropriate solutions because the efficacy of the various architectures changes as the parametric values are modified.

GNARL allows multiple structural manipulations to a network within a single mutation. As discussed earlier, constructive and destructive algorithms define a unit of modification, e.g., "add a fully connected hidden node." Because such singular structural modifications create a "one-unit structural horizon" beyond which no information is available, such algorithms may easily fixate on an architecture that is better than networks one modification step away, but worse than those two or more steps distant. In GNARL, several nodes and links can be added or deleted with each mutation, the range being determined by user-specified limits and the current ability of the network. This simultaneous modification of the structural and parametric modifications based on fitness allows the algorithm to discover appropriate networks quickly especially in comparison to evolutionary techniques that do not respect the uniqueness of distributed representations.

Finally, as in all evolutionary computation, GNARL maintains a population of structures during the search. This allows the algorithm to investigate several differing architectures in parallel while avoiding over-commitment to a particular network topology.

These search dynamics, combined with GNARL's minimal representational constraints make the algorithm extremely versatile. Of course, if topological constraints are known a priori, they should be incorporated into the search. *But these should be introduced as part of the task specification rather than being built into the search algorithm.* Because the only requirement on a fitness function f is that $f: S \rightarrow R$, diverse criteria can be used to rate a network's performance. For instance, the first two experiments described above evaluated networks based on a desired input/output mapping; the Tracker task experiment, however, considered overall network performance, not specific mappings. Other criteria could also be introduced, including specific structural constraints (e.g., minimal number of hidden units or links) as well as constraints on generalization. In some cases, strong task restrictions can even be implicit in simple fitness functions [48].

The dynamics of the algorithms guided by the task constraints represented in the fitness function allow GNARL to empirically determine an appropriate architecture. Over time, the continual cycle of *test-prune-reproduce* will constrain the population to only those architectures that have acquired the task most rapidly. Inappropriate networks will not be indefinitely competitive and will be removed from the population eventually.

Complete network induction must be approached with respect to the complex interaction between network topology, parametric values, and task performance. By fixing topology, gradient descent methods can be used to discover appropriate solutions. But the relationship between network structure and task performance is not well understood, and there is no "backpropagation" through the space of network architectures. Instead, the network induction problem is approached with heuristics that, as described above, often restrict the available architectures, the dynamics of the search mechanism, or both. Artificial architectural constraints (such as "feedforwardness") or overly constrained search mechanisms can impede the induction of entire classes of behaviors,

while forced structural liberties (such as assumed full recurrence) may unnecessarily increase structural complexity or learning time. By relying on a simple stochastic process, GNARL strikes a middle ground between these two extremes, allowing the network's complexity and behavior to emerge in response to the requirements of the task.

6.0 Acknowledgments

This research has been partially supported by ONR grants N00014-92-J-1195 and N00014-93-1-0059. We are indebted to Ed Large, Dave Stucki and especially John Kolen for proofreading help and discussions during the development of this research. Finally, we would like to thank our anonymous reviewers, and the attendees of Connectfest '92 for feedback on a preliminary versions of this work.

7.0 References

[1] A. G. Barto. Connectionist learning for control. In W. T. Miller III, R. S. Sutton, and P. J. Werbos, editors, *Neural Networks for Control*, chapter 1, pages 5–58. MIT Press, Cambridge, 1990.

[2] T. Ash. Dynamic node creation in backpropagation networks. *Connection Science*, 1(4):365–375, 1989.

[3] M. Frean. The upstart algorithm: A method for constructing and training feed-forward neural networks. Technical Report Preprint 89/469, Edinburgh Physics Dept, 1990.

[4] S. J. Hanson. Meiosis networks. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 533–541. Morgan Kaufmann, San Mateo, CA, 1990.

[5] S. E. Fahlman and C. Lebiere. The cascade-correlation architecture. In D. S. Touretsky, editor, *Advances in Neural Information Processing Structures 2*, pages 524–532. Morgan Kaufmann, San Mateo, CA, 1990.

[6] S. Fahlman. The recurrent cascade-correlation architecture. In R. Lippmann, J. Moody, and D. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 190–196. Morgan Kaufmann, San Mateo, CA, 1991.

[7] D. Chen, C. Giles, G. Sun, H. Chen, Y. Less, and M. Goudreau. Constructive learning of recurrent neural networks. *IEEE International Conference on Neural Networks*, 3:1196–1201, 1993.

[8] M. R. Azimi-Sadjadi, S. Sheedvash, and F. O. Trujillo. Recursive dynamic node creation in multilayer neural networks. *IEEE Transactions on Neural Networks*, 4(2):242–256, 1993.

[9] M. Mozer and P. Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 107–115. Morgan Kaufmann, San Mateo, CA, 1989.

[10] Y. L. Cun, J. Denker, and S. Solla. Optimal brain damage. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*. Morgan Kaufmann, San Mateo, CA, 1990.

[11] B. Hassibi and D. G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 164–171. Morgan Kaufmann, San Mateo, CA, 1993.

[12] C. W. Omlin and C. L. Giles. Pruning recurrent neural networks for improved generalization performance. Technical Report Tech Report No 93-6, Computer Science Department, Rensselaer Polytechnic Institute, April 1993.

[13] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, 1966.

[14] D. B. Fogel. *Evolving Artificial Intelligence*. Ph.D. thesis, University of California, San Diego, 1992.

[15] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1975.

[16] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1989.

[17] D. B. Fogel. An introduction to simulated evolutionary optimization. This issue.

[18] A. P. Wieland. Evolving neural network controllers for unstable systems. In *IEEE International Joint Conference on Neural Networks*, pages II-667 – II-673, IEEE Press, Seattle, WA, 1990.

[19] D. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 762–767, Morgan Kaufmann, San Mateo, CA, 1989.

[20] D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing*, 14:347–361, 1990.

[21] R. D. Beer and J. C. Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1(1):91–122, 1992.

[22] G. F. Miller, P. M. Todd, and S. U. Hegde. Designing neural networks using genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 379–384. Morgan Kaufmann, San Mateo, CA, 1989.

[23] R. K. Belew, J. McInerney, and N. N. Schraudolf. Evolving networks: Using the genetic algorithm with connectionist learning. Technical Report CS90-174, University of California, San Diego, June 1990.

[24] J. Torreele. Temporal processing with recurrent networks: An evolutionary approach. In R. K. Belew and L. B. Booker, editors, *Fourth International Conference on Genetic Algorithms*, pages 555–561. Morgan Kaufmann, San Mateo, California, 1991.

[25] M. A. Potter. A genetic cascade-correlation learning algorithm. In *Proceedings of COGANN-92 International Workshop on Combinations of Genetic Algorithms and Neural Networks*, 1992.

[26] N. Karunanithi, R. Das, and D. Whitley. Genetic cascade learning for neural networks. In *Proceedings of COGANN-92 International Workshop on Combinations of Genetic Algorithms and Neural Networks*, 1992.

[27] D. E. Goldberg. Genetic algorithms and Walsh functions: Part 2, Deception and its analysis. *Complex Systems*, 3:153–171, 1989.

[28] D. E. Goldberg. Genetic algorithms and Walsh functions: Part 1, A gentle introduction. *Complex Systems*, 3:129–152, 1989.

[29] J. D. Schaffer, D. Whitley, and L. J. Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Proceedings of COGANN-92 International Workshop on Combinations of Genetic Algorithms and Neural Networks*, 1992.

[30] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart. Distributed representations. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1: Foundations, pages 77–109. MIT Press, Cambridge, MA, 1986.

[31] T. J. Sejnowski and C. R. Rosenberg. Parallel networks that learn to pronounce english text. *Complex Systems*, 1:145–168, 1987.

[32] J. Koza and J. Rice. Genetic generation of both the weights and architecture for a neural network. In *IEEE International Joint Conference on Neural Networks*, pages II-397 – II-404, Seattle, WA, IEEE Press, 1991.

[33] R. Collins and D. Jefferson. An artificial neural network representation for artificial organisms. In H. P. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*. Springer-Verlag, 1991.

[34] D. B. Fogel. A brief history of simulated evolution. In D. B. Fogel and W. Atmar, editors, *Proceedings of the First Annual Conference on Evolutionary Programming*, Evolutionary Programming Society, La Jolla, CA., 1992.

[35] D. B. Fogel, L. J. Fogel, and V. W. Porto. Evolving neural networks. *Biological Cybernetics*, 63:487–493, 1990.

[36] J. R. McDonnell and D. Waagen. Determining neural network connectivity using evolutionary programming. In *Twenty-fifth Asilomar Conferences on Signals, Systems, and Computers*, Monterey, CA, 1992.

[37] D. B. Fogel. Using evolutionary programming to create neural networks that are capable of playing Tic-Tac-Toe. In *International Conference on Neural Networks*, pages 875–880. IEEE Press, San Francisco, CA, 1993.

[38] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[39] R. J. Williams. *Adaptive State Representation and Estimation Using Recurrent Connectionist Networks*, chapter 4, pages 97–114. MIT Press, Cambridge, MA, 1990.

[40] J. B. Pollack. The induction of dynamical recognizers. *Machine Learning*, 7:227–252, 1991.

[41] M. Tomita. Dynamic construction of finite automata from examples using hill-climbing. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, pages 105–108, Ann Arbor, MI, 1982.

[42] R. L. Watrous and G. M. Kuhn. Induction of finite-state automata using second-order recurrent networks. In *Advances in Neural Information Processing 4*. Morgan Kaufmann, San Mateo, CA, 1992.

[43] C. L. Giles, G. Z. Sun, H. H. Chen, Y. C. Lee, and D. Chen. Higher order recurrent networks & grammatical inference. In D. S. Touretsky, editor, *Advances in Neural Information Processing Systems 2*, pages 380-387. Morgan Kaufmann, San Mateo, CA, 1990.

[44] C. L. Giles, C. B. Miller, D. Chen, G. Z. Sun, H. H. Chen, and Y. C. Lee. Extracting and learning an unknown grammar with recurrent neural networks. In *Advances in Neural Information Processing 4*. Morgan Kaufmann, San Mateo, CA, 1992.

[45] Z. Zeng, R. M. Goodman, and P. Smyth. Learning finite state machines with self-clustering recurrent networks. *Neural Computation*, to appear.

[46] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang. Evolution as a theme in artificial life: The genesys/tracker system. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II: Proceedings of the Workshop on Artificial Life*, pages 549–577. Addison-Wesley, 1991.

[47] J. Koza. Genetic evolution and co-evolution of computer programs. In J. D. F. Christopher G. Langton, Charles Taylor and S. Rasmussen, editors, *Artificial Life II*. Addison Wesley Publishing Company, Reading Mass., 1992.

[48] P. J. Angeline and J. B. Pollack. Competitive environments evolve better solutions for complex tasks. In S. Forrest, editor, *Genetic Algorithms: Proceedings of the Fifth International Conference (GA93)*, Morgan Kaufmann, San Mateo, CA, 1993.